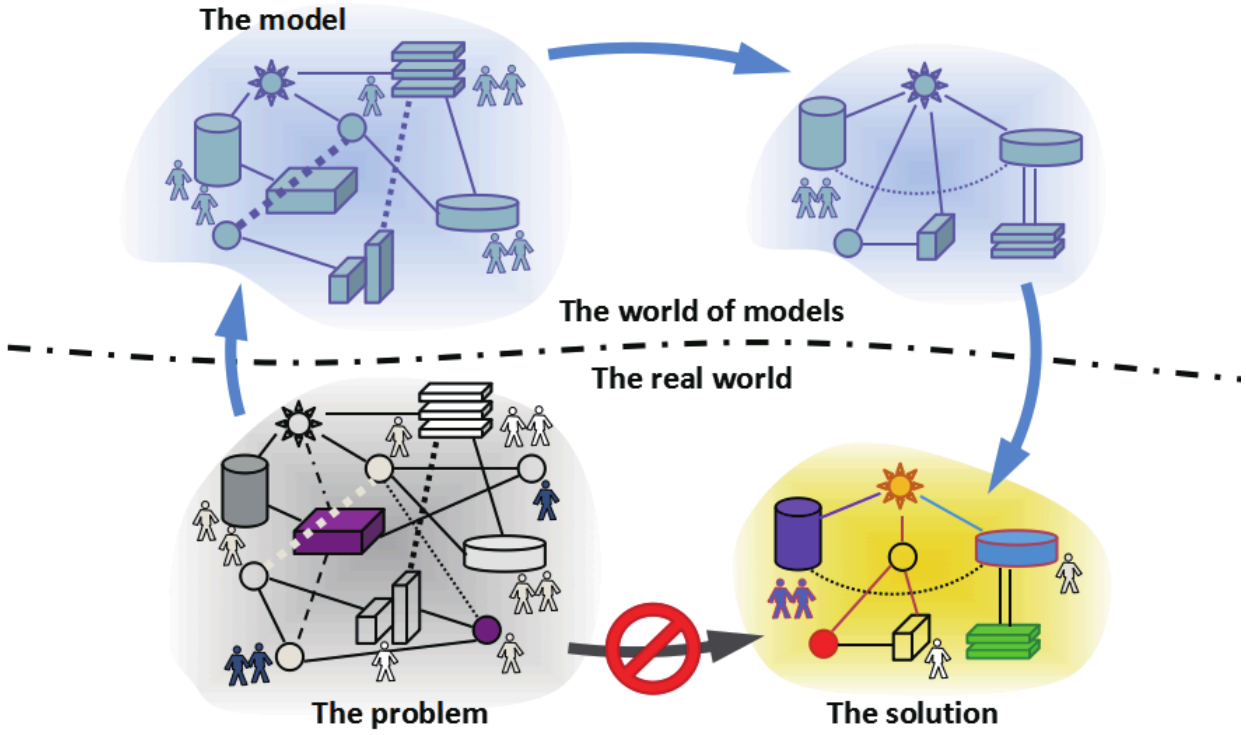


The solution at the model level

ANDREI BORSHCHEV

The Big Book of

Simulation Modeling

Multimethod Modeling with Anylogic 6

ANDREI BORSHCHEV

The Big Book of Simulation Modeling

Multimethod Modeling with AnyLogic 6

AnyLogic North America
2013

ISBN-13: 978-0-9895731-7-7

ISBN-10: 0989573176

© Copyright 2013 AnyLogic North America. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

About the Author

Dr. Andrei Borshchev is the CEO and co-founder of The AnyLogic Company, a leading provider of dynamic simulation tools, technologies, and consulting services for business applications, and is personally recognized as one of the world's top experts in Agent Based and multi-method simulation modeling.

Since 1998, Dr. Borshchev has been leading the development of AnyLogic software enabling it to be considered the corporate standard for simulation in over 500 organizations worldwide. He has conducted hundreds of lectures, seminars, and training sessions on simulation modeling and AnyLogic in the USA, Canada, Europe, Russia, Australia, China, Korea, Japan, and South Africa, and has also led numerous simulation based consulting projects across multiple markets including logistics, transportation, supply chains, defense, healthcare, manufacturing, and consumer market analysis.

Preface

This book is for modelers.

There are great books on using probability distributions in simulation models, on statistical analysis of the model output, on model calibration, on design of Monte Carlo experiments, on performing sensitivity analysis. In other words, things that are *around* the simulation model (and are, of course, essential parts of a simulation modeling project) are well covered in the literature.

The actual *creation* of simulation models, i.e. the process of finding the right abstraction level, choosing the right modeling language, and using the right constructs to create a representation of the problem suitable for dynamic experiments, has been treated like "more art than technology", and is not covered well enough. The existing books typically assume a certain approach in the very beginning and then present a set of examples with growing complexity.

The books from the "Discrete Event simulation camp" would say: "start with the most high-level representation of the system, build the simplest process model, and then go down and add details to the process components as necessary". Well, starting with the simplest thing possible is definitely a fundamental principle of any simulation project, but is the "simple process model" really the most high-level representation of the system? How about a System Dynamics model, where individual entities and resources are not even present? Maybe that is a lot better starting point?

On the other hand, have you seen the system dynamics "monster models" with hundreds and thousands of variables, sophisticated subscripts and delay constructs? In many cases, a good part of that complexity might go away, should the modeler, during one of the iterations, decide to represent a part of the system using a discrete event or agent based paradigm. (To be fair, we should say that the "system dynamics camp" literature, while staying within one modeling paradigm, treats the choice of the level of abstraction and the model boundary very seriously; the author considers the book (Sterman, 2000) as the best book on modeling ever written.)

System dynamics and discrete event (process-centric) modeling are the two traditional modeling paradigms with over 50 years history. They existed and were developing (or not developing) in almost complete isolation one from another: different communities, different conferences, different books, and different software tools with zero interoperability.

By the early 2000s the third powerful modeling paradigm had arrived: Agent Based modeling. It has its roots in computer science, in particular, in object-oriented modeling. Agent based modeling suggests a completely different view the modeler should take when mapping a real world system to its virtual representation. It also offers great new languages for describing dynamics, in particular the language of Statecharts. Agent based modeling has already found its place in the simulation practitioner's toolkit; however, there is a significant lack of practically useful texts on the topic.

***

The author is a strong adopter of minimalist philosophy in modeling, the Occam's razor principle in particular. Another thing the author believes in is that style and beauty are important. The model that is not clean and beautiful internally is not a good model. Workarounds (unnatural constructs) are always ugly. And the source of workarounds in many cases is the inability to express the real world phenomenon within a chosen modeling language at the necessary level of abstraction.

I have seen many models that could become times smaller, a lot more manageable, faster, and cleaner, had the modeler used a different method, or a different language. People try to fit interacting individual behaviors into the process paradigm, emulate processes with stocks, flows, and delay structures, and so on, not to speak about spreadsheets wrongly used instead of simulation tools (and vice versa!).

The key to creation of clean, minimalistic, manageable, and workaround-free models where modeling languages are used *naturally* and the constructs are beautiful is *knowledge of different methods and languages and wider thinking*. This book is an overview of all three methods, or paradigms, in simulation modeling combined with a practical guide to model building. The book is based on AnyLogic, the software tool that enables a modeler to utilize all three methods and to combine them in a single model. More than 100 step-by-step examples given in the book are taken from our 15 years experience of building commercial simulation models and supporting thousands of AnyLogic users, simulation practitioners.

***

The first edition of the book focuses on agent based, system dynamics, and multi-method modeling; discrete event models are mostly presented in combination with other methods. The book includes a detailed introduction to statecharts – one of the primary languages for agent based modeling. It also covers design of interactive 2D and 3D animation, and some technical aspects of model building, in particular, Java basics for modelers and data exchange with external programs. One of the AnyLogic domain-specific libraries, the Rail Library, is described here as well. The second

edition of the book is already being written. It will include in-depth chapters on discrete event modeling, advanced system dynamics, experiment design, and pedestrian flow modeling.

The AnyLogic software is not included with the book, but can be downloaded at www.anylogic.com/downloads. Almost all example models used in the book are packaged with the software version 6.9, the few that are not can be found at the online portal www.RunTheModel.com.

***

I would like to thank:

Prof. **Yuri Karpov** (St.Petersburg State Polytechnic University) and **Alexei Filippov** (now Google) who were with me during the initial stages of AnyLogic design and development. The concept of the great picture that you can see on the book cover also belongs to Yuri Karpov.

Lyle Wallis (Decisio Consulting) and **Mark Paich** (Decisio Consulting, Lexidyne, and PricewaterhouseCoopers) who patiently helped us to shape AnyLogic as a multi-method simulation platform and actually introduced us to system dynamics and agent based modeling.

Chris Johnson and his team at GE Global Research, in particular **Jens Alkemper** and **Onur Dulgeroglu**, for their continuous feedback and ideas on how we can improve AnyLogic.

Geoff McDonnell (University of New South Wales and Adaptive Care Systems), **Nate Osgood** (University of Saskatchewan), and **Stefan Bengtsson** (Stockholms Läns Landsting) for bringing AnyLogic and multi-method modeling into the health care area.

The AnyLogic team members who helped me in various ways while I was writing the book: **Alena Beloshapko, Nikolay Churkov, Maxim Garifullin** (now GE Healthcare), **Irina Gleim, Victor Gleim** (now Luxoft), **Elena Grigoryeva, Sergey Iyust, Vladimir Koltchanov** (AnyLogic Europe), **Pavel Lebedev, Ivan Lelekov, George Meringov** (now freelance software developer), **Andrei Nozhenko, Timofey Popkov** (he actually insisted that I write this book), **Nikolay Rassadin** (now freelance developer), **Sergey Suslov, Yulia Tropina, Dmitry Vrubel, Anatoly Zherebtsov**.

And especially **Ilya Grigoryev** for his great impact on the quality of the book and preparing the book for publication.

I would also like to thank thousands of AnyLogic users worldwide who build great models and give us their feedback on how we can further improve the exciting multi-method simulation modeling technology.

Contents

CHAPTER 1. MODELING AND SIMULATION MODELING 25

1.1. TYPES OF MODELS 26
1.2. ANALYTICAL VS. SIMULATION MODELING 27
The limits of analytical modeling: queuing theory 29
Advantages of simulation modeling 34
1.3. APPLICATIONS OF SIMULATION MODELING. LEVEL OF ABSTRACTION. METHODS 34

CHAPTER 2. THE THREE METHODS IN SIMULATION MODELING 37

2.1. SYSTEM DYNAMICS 38
Example 2.1: New product diffusion 39
Underlying mathematics and simulation engine 44
Abstraction level 44
Software tools 45
2.2. DISCRETE EVENT MODELING 45
Example 2.2: Bank 46
Abstraction level 47
Underlying mathematics and simulation engine 48
Software tools 49
2.3. AGENT BASED MODELING 49
Example 2.3: Agent based epidemic model 50
Abstraction level 54
For those who have read books and papers on agent based modeling 54
Underlying mathematics and simulation engine 55
Software tools 55

CHAPTER 3. AGENT BASED MODELING. TECHNOLOGY OVERVIEW 57

3.1. WHO ARE THE AGENTS? 57
Who are the agents in an American automotive market model? 58
3.2. AGENT BASED MODELING AND OBJECT-ORIENTED DESIGN 60
OO modeling in AnyLogic 63
3.3. TIME IN AGENT BASED MODELS 67
3.4. SPACE IN AGENT BASED MODELS 69
3.5. DISCRETE SPACE 71
Example 3.1: Schelling segregation 74
Example 3.2: Conway's Game of Life 76
Example 3.3: Wildfire 78

Discrete space API *85*
3.6. CONTINUOUS 2D AND 3D SPACE 86
Movement in continuous space *88*
Example 3.4: Air defense system *90*
Example 3.5: Agent leaving a movement trail *105*
Continuous space API *109*
3.7. NETWORKS AND LINKS 110
Standard networks *111*
Example 3.6: Periodic repair of a standard network *113*
Example 3.7: Custom network built using standard connections *116*
Fully connected networks *118*
Network and layout-related API *118*
Unidirectional, temporary, and other custom types of links *119*
Example 3.8: Kinship modeled using custom links *122*
A note on vertical links in hierarchical models *127*
Using ports to connect agents *127*
3.8. COMMUNICATION BETWEEN AGENTS. MESSAGE PASSING 128
Synchronous and asynchronous communication *128*
API for message passing *129*
Message handling *131*
Other types of inter-agent communication *131*
3.9. DYNAMIC CREATION AND DESTRUCTION OF AGENTS 132
3.10. STATISTICS ON AGENT POPULATIONS 134
Example 3.9: Kinship model with standard statistics *134*
Example 3.10: Kinship model with dynamic histograms *136*
Customized high performance statistics *138*
Example 3.11: Kinship model with customized statistics *139*
3.11. CONDITION-TRIGGERED EVENTS AND TRANSITIONS IN AGENTS 140

CHAPTER 4. HOW TO BUILD AGENT BASED MODELS. FIELD SERVICE EXAMPLE 142

4.1. THE PROBLEM STATEMENT 143
4.2. PHASE 1. CAN BE DONE ON PAPER 144
Who are the agents? *144*
Equipment unit agent *145*
Service crew agent *147*
Agent communication. Message sequence diagrams *150*
Space and other things shared by all agents *153*
4.3. PHASE 2. THE MODEL IN ANYLOGIC. THE FIRST RUN 154
The model structure and the top level object Main *154*

The EquipmentUnit agent ... *155*
The ServiceCrew agent ... *158*
Animation ... *159*
The first run ... *160*
Discussion and next steps ... *161*
4.4. PHASE 3. THE MISSING FUNCTIONALITY ... 162
Maintenance, age, and failure rate ... *162*
Scheduling maintenance. Handling requests of two types ... *164*
Discussion. Code in the model ... *167*
4.5. PHASE 4. MODEL OUTPUT. STATISTICS. COST AND REVENUE CALCULATION ... 169
Equipment availability and service crew utilization ... *169*
Cost and revenue ... *174*
4.6. PHASE 5. CONTROL PANEL. RUNNING THE FLIGHT SIMULATOR ... 177
Design of control panel ... *177*
Changing the number of service crews ... *178*
Equipment replacement policy ... *179*
Running the flight simulator ... *180*
4.7. PHASE 6. USING THE OPTIMIZER TO FIND THE BEST SOLUTION ... 182
Preparing the model for optimization ... *182*
Setting up the optimization experiment ... *183*
Optimization run ... *185*
4.8. ASSUMPTIONS ... 187
4.9. BONUS PHASE. 3D ANIMATION ... 188
4.10. BONUS DISCUSSION. COULD WE MODEL THIS IN DISCRETE EVENT STYLE? ... 191

CHAPTER 5. SYSTEM DYNAMICS AND DYNAMIC SYSTEMS ... 195

5.1. HOW TO DRAW STOCK AND FLOW DIAGRAMS ... 196
Drawing stocks and flows ... *196*
Drawing variables, dependency links, polarities, and loop types ... *197*
Naming conventions for system dynamics variables ... *199*
Layout of large models. "Sectors" and shadow variables ... *199*
5.2. EQUATIONS ... 202
Using Java in SD equations ... *204*
"Constant variables" and parameters ... *205*
Units and unit checking ... *206*
5.3. EXAMPLE: POPULATION AND CARRYING CAPACITY ... 208
Phase 1: Unlimited resources. Positive feedback. Exponential growth ... *208*
Customizing the dataset collection ... *212*
Phase 2: Crowding affects lifetime. Negative feedback. S-shaped growth ... *213*

Phase 3: Crowding affects births 216
Phase 4: Negative feedback with delay. Overshoot and oscillation 218
Specifying units and performing unit checking 220
5.4. OTHER TYPES OF EXPERIMENTS. INTERACTIVE GAMES 221
Example 5.1: New product diffusion - compare runs 221
Example 5.2: New product diffusion - sensitivity analysis 224
Example 5.3: Epidemic model – calibration 225
Example 5.4: Epidemic model - instant charts 229
Example 5.5: Stock management game 232
5.5. EXPORTING THE MODEL AND PUBLISHING IT ON THE WEB 237

CHAPTER 6. MULTI-METHOD MODELING 240

6.1. ARCHITECTURES 241
The choice of model architecture and methods 243
6.2. TECHNICAL ASPECT OF COMBINING MODELING METHODS 244
Examples 5.1 - 5.21: Combining modeling methods 244
System dynamics -> discrete elements 244
Discrete elements -> system dynamics 246
Agent based <-> discrete event 253
Referencing model elements located in different active objects 257
The simulation performance of multi-method models 259
6.3. EXAMPLES 259
Example 6.22: Epidemic and clinic 259
Example 6.23: Consumer market and supply chain 267
Example 6.24: Product portfolio and investment policy 272
6.4. DISCUSSION 285

CHAPTER 7. DESIGNING STATE-BASED BEHAVIOR: STATECHARTS 287

7.1. WHAT IS A STATECHART? 287
Example 7.1: A laptop running on a battery 287
How do statecharts differ from action charts and flowcharts? 288
7.2. DRAWING STATECHARTS 289
Simple states 290
Transitions 290
Statechart entry point 291
Composite states 292
History state 293
Final state 294
7.3. STATE TRANSITIONS: TRIGGERS, GUARDS, AND ACTIONS 294

Which transitions are active? ... 294
Trigger types ... 295
Timeout expressions ... 296
Transitions triggered by messages ... 297
Sending messages to a statechart ... 299
Guards of transitions ... 300
Transitions with branches ... 301
Internal transitions ... 302
Order of action execution ... 303
Synchronous vs. asynchronous transitions ... 304
7.4. STATECHART-RELATED API ... 306
7.5. VIEWING AND DEBUGGING THE STATECHARTS AT RUNTIME ... 307
7.6. STATECHARTS FOR PEOPLE'S LIVES AND BEHAVIOR ... 308
Example 7.2: Life phases ... 308
Example 7.3: Adoption and diffusion ... 309
Example 7.4: Disease diffusion ... 310
Example 7.5: Purchase behavior with a choice of two competing products ... 311
7.7. STATECHARTS FOR PHYSICAL OBJECTS ... 313
Example 7.6: Generic resource with breakdowns and repairs ... 313
Example 7.7: Delivery truck ... 314
Example 7.8: Aircraft maintenance checks ... 314
7.8. STATECHARTS FOR PRODUCTS AND PROJECTS ... 316
Example 7.9: Product life cycle, including NPD ... 316
Example 7.10: Pharmaceutical NPD pipeline ... 318
7.9. STATECHARTS FOR TIMING ... 319
Example 7.11: Statechart for shop working hours ... 319

CHAPTER 8. DISCRETE EVENTS AND EVENT MODEL OBJECT ... 321

8.1. DISCRETE EVENTS ... 321
The terminology ... 321
Discrete events: approximation of real world continuous processes ... 321
Discrete event management inside AnyLogic engine ... 323
8.2. EVENT – THE SIMPLEST LOW LEVEL MODEL OBJECT ... 324
Example 8.1: Event writes to the model log every time unit ... 326
Example 8.2: Event generates new agents ... 328
Events triggered by a condition ... 329
Example 8.3: Event waits on a stock reaching a certain level ... 330
Example 8.4: Automatic shutdown after a period of inactivity ... 331
Example 8.5: Event slows down the simulation on a particular date ... 333

Event API *334*
8.3. DYNAMIC EVENTS 335
Example 8.6: Product delivery *336*
API related to dynamic events *338*

CHAPTER 9. RAILS AND TRAINS **340**

9.1. DEFINING THE RAIL TOPOLOGY 341
Example 9.1: A very simple rail yard *342*
3D animation of rail yards *345*
Creating rail yards programmatically *345*
Example 9.2: Creating a rail yard by code *346*
Java class Track *347*
Java class Switch *349*
9.2. DEFINING THE OPERATION LOGIC OF THE RAIL MODEL 350
Example 9.3: Train stop *352*
Example 9.4: Ensuring safe movement of trains *357*
Example 9.5: Simple classification yard *363*
Example 9.6: Airport shuttle train (featuring AnyLogic Pedestrian Library) *372*
Java class Train (subclass of Entity) *375*
Java class RailCar *377*

CHAPTER 10. JAVA FOR ANYLOGIC USERS **380**

10.1. PRIMITIVE DATA TYPES 381
10.2. CLASSES 382
Class as grouping of data and methods. Objects as instances of class *382*
Inheritance. Subclass and super class *383*
Classes and objects in AnyLogic models *385*
10.3. VARIABLES (LOCAL VARIABLES AND CLASS FIELDS) 387
Local (temporary) variables *387*
Class variables (fields) *388*
10.4. FUNCTIONS (METHODS) 389
Standard and system functions *390*
Functions of the model elements *392*
Defining your own function *394*
10.5. EXPRESSIONS 396
Arithmetic expressions *396*
Relations and equality *398*
Logical expressions *398*
String expressions *399*

Conditional operator ?: ... *400*
10.6. JAVA ARRAYS AND COLLECTIONS ... 400
Arrays ... *401*
Collections ... *403*
Replicated active objects are collections too ... *407*
10.7. NAMING CONVENTIONS ... 408
10.8. STATEMENTS ... 410
Variable declaration ... *411*
Function call ... *412*
Assignment ... *412*
If-then-else ... *413*
Switch ... *414*
For loop ... *415*
While loop ... *416*
Block {...} and indentation ... *417*
Return statement ... *418*
Comments ... *419*
10.9. WHERE AM I AND HOW DO I GET TO...? ... 421
10.10. VIEWING JAVA CODE GENERATED BY ANYLOGIC ... 423
10.11. CREATING YOUR JAVA CLASSES WITHIN ANYLOGIC MODEL ... 425
Inner classes ... *428*
10.12. LINKING EXTERNAL JAVA MODULES (JAR FILES) ... 429

CHAPTER 11. EXCHANGING DATA WITH EXTERNAL WORLD ... 430

11.1. TEXT FILES ... 431
Example 11.1: Using text file as a log ... *432*
Example 11.2: Reading table function from a text file ... *433*
Example 11.3: Reading agent parameters from a CSV file ... *435*
11.2. EXCEL SPREADSHEETS ... 437
Example 11.4: Reading data of various types from fixed cells in Excel ... *439*
Example 11.5: Reading model parameters from Excel using Java reflection ... *441*
Example 11.6: Displaying the model output as a chart in Excel ... *443*
11.3. DATABASES ... 446
SQL queries ... *448*
AnyLogic database connectivity objects ... *451*
Example 11.7: Loading data from a database and using ResultSet ... *453*
Example 11.8: Creating agent populations parameterized from a database ... *455*
Example 11.9: Dumping simulation output into a database table ... *459*
Example 11.10: Using prepared statement when writing to databases ... *462*

11.4. WORKING WITH THE CLIPBOARD 464
Example 11.11: Working with clipboard 464
11.5. STANDARD OUTPUT, THE MODEL LOG, AND COMMAND LINE ARGUMENTS 466

CHAPTER 12. PRESENTATION AND ANIMATION: WORKING WITH SHAPES, GROUPS, COLORS 468

12.1. DRAWING AND EDITING SHAPES 469
Polylines and curves 470
Arcs 471
Text 471
Images 472
Z-Order 474
Selecting hidden shapes 475
Coordinates and the grid 475
Copying shapes 477
Locking shapes – preventing selection by mouse 477
General properties of graphical shapes 478
Advanced properties of graphical shapes 479
12.2. GROUPING SHAPES 480
3D Groups 485
Working with the group contents dynamically using API 485
On draw extension point – execute custom code on each frame 486
Groups in the project tree 486
Top level groups for active object presentation and icon 487
12.3. ANIMATION PRINCIPLES. DYNAMIC PROPERTIES OF SHAPES 487
Dynamic properties of shapes 488
Example 12.1: Commodity price change animation 489
Example 12.2: Elevator doors animation 491
Example 12.3: Stock of money animation 493
Example 12.4: Missile attack animation 495
Animation frames 497
12.4. REPLICATED SHAPES 498
Example 12.5: Drawing seats in a movie theater 499
Example 12.6: Selling seats in the movie theater 501
Example 12.7: Drawing a flower 502
Example 12.8: Product portfolio bubble chart (BCG chart) 504
12.5. SHAPES' API 507
Example 12.9: Using color to show the current state of a statechart 507
Example 12.10: Show/hide a callout 508

Example 12.11: Read graphics from a text file 510
Example 12.12: Find all red circles 512
Example 12.13: Resize the red circles 513
API of non-persistent shapes 514
AnyLogic Java class hierarchy for shapes 514
12.6. COLORS AND TEXTURES 514
Example 12.14: Choosing appropriate colors for an arbitrary number of objects. 516
Transparency 517
Example 12.15: Using transparency to show coverage zone 517
Example 12.16: Show population density using color interpolation 518

CHAPTER 13. DESIGNING INTERACTIVE MODELS: USING CONTROLS 521

Example 13.1: Slider linked to a model parameter 522
Example 13.2: Buttons changing the parameter value 523
Example 13.3: Edit box linked to a parameter of embedded object 525
Example 13.4: Radio buttons changing the view mode 526
Example 13.5: Combo box controlling the simulation speed 527
Example 13.6: File chooser for text files 529
Indivisibility of control actions and model events 530
13.1. DYNAMIC PROPERTIES OF CONTROLS 530
Example 13.7: Radio buttons enabling/disabling other controls 530
Example 13.8: Keeping controls in the top left corner of the window 531
Example 13.9: Replicated button 532
13.2. CONTROLS' API 533
13.3. HANDLING MOUSE CLICKS 534
Example 13.10: Hyper link menu to navigate between view areas 535
Example 13.11: Creating dots at the click coordinates 536
Example 13.12: Catching mouse clicks anywhere on the canvas 537

CHAPTER 14. 3D ANIMATION 539

Example 14.1: A very simple model with 3D animation 540
14.1. PRIMITIVE 3D SHAPES 542
14.2. 3D GROUPS AND ROTATION 545
Example 14.2: Rotation in 3D – a sign on two posts 545
Example 14.3: Bridge crane 3D 546
14.3. STANDARD AND IMPORTED 3D GRAPHICS 549
Using standard 3D graphics 549
Using external 3D graphics 550
14.4. HIERARCHICAL 3D ANIMATIONS. EMBEDDED 3D PRESENTATION 551

14.5. 3D WINDOWS 552
Navigation in the 3D scene at runtime 552
Multiple 3D views 553
14.6. CAMERAS 554
Example 14.4: A very simple model with multiple 3D windows and cameras 554
Example 14.5: Camera on a moving object 556
14.7. LIGHTS 559
Example 14.6: Examples of Lights in 3D Scene 561

CHAPTER 15. RANDOMNESS IN ANYLOGIC MODELS 565

15.1. PROBABILITY DISTRIBUTIONS 565
Probability distribution functions 567
Distribution fitting 570
Custom (empirical) distributions 570
15.2. SOURCES OF RANDOMNESS IN THE MODEL 573
Randomness in process models 574
Randomness in agent based models 576
Example 15.1: Agents randomly distributed within a freeform area 578
Example 15.2: Agents randomly distributed over a finite set of locations 579
Randomness in system dynamics models 581
Example 15.3: Stock price fluctuations in a system dynamics model 581
Randomness in AnyLogic simulation engine 582
15.3. RANDOM NUMBER GENERATORS. REPRODUCIBLE AND UNIQUE EXPERIMENTS 583
Random number generators 583
The seed. Reproducible and unique experiments 585
Example 15.4: Reproducible experiment with a stochastic process model 585

CHAPTER 16. MODEL TIME, DATE AND CALENDAR. VIRTUAL AND REAL TIME 589

16.1. THE MODEL TIME 589
Time units 590
Developing models independent of time unit settings 591
16.2. DATE AND CALENDAR 592
Finding out the current date, day of week, hour of day, etc. 594
Constructing dates. Converting the model date to the model time and vice versa 595
Specifying timeouts and delays in days, months, years 596
16.3. VIRTUAL AND REAL-TIME EXECUTION MODES 597
Execution mode API 599

REFERENCES 601

INDEX 605

List of examples

Example 2.1: New product diffusion 39
Example 2.2: Bank 46
Example 2.3: Agent based epidemic model 50
Example 3.1: Schelling segregation 74
Example 3.2: Conway's Game of Life 76
Example 3.3: Wildfire 78
Example 3.4: Air defense system 90
Example 3.5: Agent leaving a movement trail 105
Example 3.6: Periodic repair of a standard network 113
Example 3.7: Custom network built using standard connections 116
Example 3.8: Kinship modeled using custom links 122
Example 3.9: Kinship model with standard statistics 134
Example 3.10: Kinship model with dynamic histograms 136
Example 3.11: Kinship model with customized statistics 139
Example 5.1: New product diffusion - compare runs 221
Example 5.2: New product diffusion - sensitivity analysis 224
Example 5.3: Epidemic model – calibration 225
Example 5.4: Epidemic model - instant charts 229
Example 5.5: Stock management game 232
Examples 5.1 - 5.21: Combining modeling methods 244
Example 6.22: Epidemic and clinic 259
Example 6.23: Consumer market and supply chain 267
Example 6.24: Product portfolio and investment policy 272
Example 7.1: A laptop running on a battery 287
Example 7.2: Life phases 308
Example 7.3: Adoption and diffusion 309
Example 7.4: Disease diffusion 310
Example 7.5: Purchase behavior with a choice of two competing products 311
Example 7.6: Generic resource with breakdowns and repairs 313
Example 7.7: Delivery truck 314
Example 7.8: Aircraft maintenance checks 314
Example 7.9: Product life cycle, including NPD 316
Example 7.10: Pharmaceutical NPD pipeline 318
Example 7.11: Statechart for shop working hours 319
Example 8.1: Event writes to the model log every time unit 326
Example 8.2: Event generates new agents 328

Example 8.3: Event waits on a stock reaching a certain level 330
Example 8.4: Automatic shutdown after a period of inactivity 331
Example 8.5: Event slows down the simulation on a particular date 333
Example 8.6: Product delivery 336
Example 9.1: A very simple rail yard 342
Example 9.2: Creating a rail yard by code 346
Example 9.3: Train stop 352
Example 9.4: Ensuring safe movement of trains 357
Example 9.5: Simple classification yard 363
Example 9.6: Airport shuttle train (featuring AnyLogic Pedestrian Library) 372
Example 11.1: Using text file as a log 432
Example 11.2: Reading table function from a text file 433
Example 11.3: Reading agent parameters from a CSV file 435
Example 11.4: Reading data of various types from fixed cells in Excel 439
Example 11.5: Reading model parameters from Excel using Java reflection 441
Example 11.6: Displaying the model output as a chart in Excel 443
Example 11.7: Loading data from a database and using ResultSet 453
Example 11.8: Creating agent populations parameterized from a database 455
Example 11.9: Dumping simulation output into a database table 459
Example 11.10: Using prepared statement when writing to databases 462
Example 11.11: Working with clipboard 464
Example 12.1: Commodity price change animation 489
Example 12.2: Elevator doors animation 491
Example 12.3: Stock of money animation 493
Example 12.4: Missile attack animation 495
Example 12.5: Drawing seats in a movie theater 499
Example 12.6: Selling seats in the movie theater 501
Example 12.7: Drawing a flower 502
Example 12.8: Product portfolio bubble chart (BCG chart) 504
Example 12.9: Using color to show the current state of a statechart 507
Example 12.10: Show/hide a callout 508
Example 12.11: Read graphics from a text file 510
Example 12.12: Find all red circles 512
Example 12.13: Resize the red circles 513
Example 12.14: Choosing appropriate colors for an arbitrary number of objects 516
Example 12.15: Using transparency to show coverage zone 517
Example 12.16: Show population density using color interpolation 518
Example 13.1: Slider linked to a model parameter 522
Example 13.2: Buttons changing the parameter value 523

EXAMPLE 13.3: EDIT BOX LINKED TO A PARAMETER OF EMBEDDED OBJECT 525
EXAMPLE 13.4: RADIO BUTTONS CHANGING THE VIEW MODE 526
EXAMPLE 13.5: COMBO BOX CONTROLLING THE SIMULATION SPEED 527
EXAMPLE 13.6: FILE CHOOSER FOR TEXT FILES 529
EXAMPLE 13.7: RADIO BUTTONS ENABLING/DISABLING OTHER CONTROLS 530
EXAMPLE 13.8: KEEPING CONTROLS IN THE TOP LEFT CORNER OF THE WINDOW 531
EXAMPLE 13.9: REPLICATED BUTTON 532
EXAMPLE 13.10: HYPER LINK MENU TO NAVIGATE BETWEEN VIEW AREAS 535
EXAMPLE 13.11: CREATING DOTS AT THE CLICK COORDINATES 536
EXAMPLE 13.12: CATCHING MOUSE CLICKS ANYWHERE ON THE CANVAS 537
EXAMPLE 14.1: A VERY SIMPLE MODEL WITH 3D ANIMATION 540
EXAMPLE 14.2: ROTATION IN 3D – A SIGN ON TWO POSTS 545
EXAMPLE 14.3: BRIDGE CRANE 3D 546
EXAMPLE 14.4: A VERY SIMPLE MODEL WITH MULTIPLE 3D WINDOWS AND CAMERAS 554
EXAMPLE 14.5: CAMERA ON A MOVING OBJECT 556
EXAMPLE 14.6: EXAMPLES OF LIGHTS IN 3D SCENE 561
EXAMPLE 15.1: AGENTS RANDOMLY DISTRIBUTED WITHIN A FREEFORM AREA 578
EXAMPLE 15.2: AGENTS RANDOMLY DISTRIBUTED OVER A FINITE SET OF LOCATIONS 579
EXAMPLE 15.3: STOCK PRICE FLUCTUATIONS IN A SYSTEM DYNAMICS MODEL 581
EXAMPLE 15.4: REPRODUCIBLE EXPERIMENT WITH A STOCHASTIC PROCESS MODEL 585

Chapter 1. Modeling and simulation modeling

In this chapter we will talk about modeling in general, types of models, and then focus on simulation modeling.

Modeling is one of the ways to solve problems that appear in the real world. In many cases we cannot afford finding the right solutions by experimenting with real objects: building, destroying, making changes may be too expensive, dangerous, or just impossible. If this is so, we leave the real world and go up to the world of models, see Figure 1.1. We build a model of a real system: its representation in a modeling language. This process assumes abstraction: we throw away the details that (we think) are irrelevant to the problem we are trying to solve and keep what we think is important. The model is always less complex than the original system.

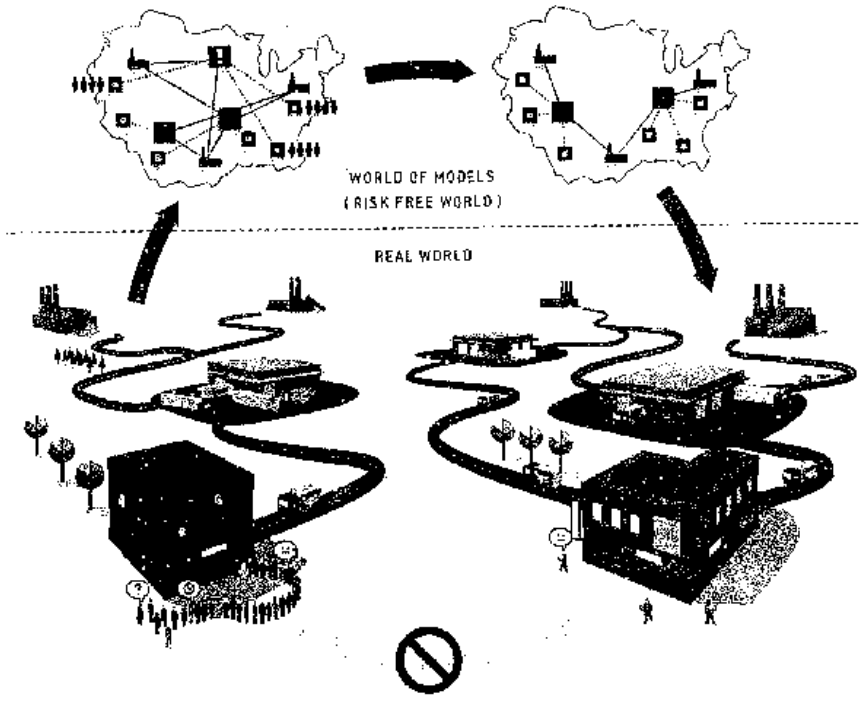


Figure 1.1 Modeling

- **The phase of building the model, that is mapping the real world to the world of models, choosing the abstraction level and the modeling language (= the method) is a less formalized thing in the whole process of using models to solve problems. This is still more an art than a science.**

Having built the model (or sometimes even while building the model), we start to explore and understand the structure and behavior of the original system, test how the system will behave under various conditions, play and compare different scenarios, optimize. When we find the solution we are looking for, we map that solution back to the real world.

- **The whole modeling thing is actually about finding the way from the problem to its solution through a risk-free world where we are allowed to make mistakes, undo things, go back in time, and start all over again.**

1.1. Types of models

There are many different types of models that we build. Consider Figure 1.2. Everybody builds mental models every day. A mental model is your understanding of how things work in the real world: friends, family, colleagues, car drivers, town where you live, things that you buy, economy, sports, politics, your own body. Decisions like what to say to your kid, what to eat for breakfast, who to vote for, or where to take your girlfriend tonight are all based on mental models.

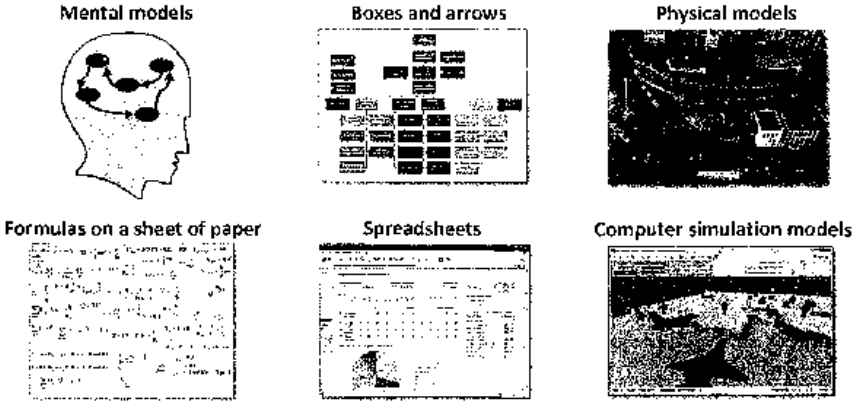


Figure 1.2 Types of models

An org chart of a company drawn using boxes with arrows is a model. You can use it to explain the structure of the organization, you can move people from one

position or group to another and think about the advantages and drawbacks of a new structure.

If you take a pen and a sheet of paper and derive a formula for the optimal cross-slope of a road on a bend as a function of the turning radius and vehicle speed, you actually create an analytical model of vehicle movement in a turn based on another model – Newton's laws of motion.

Models can be physical. A model railroad can be used, for example, to optimize the layout and operation of a classification yard. A wind tunnel is a model of a free flight used to study aerodynamic forces and optimize the shape and design of the airplane.

Computers provide us with a flexible virtual world where we can easily create anything we can imagine. They are naturally and extensively used for modeling. Computer models can be of different kinds. The spreadsheet is the most accessible modeling software where someone can model arithmetic or algebra - such as expenses, for near foreseeable future. Software such as MS Visio™ can be used, for example, to plan your office layout; Autodesk 3ds Max™ to visualize interior design; Wolfram Mathematica™ to perform fast exact-match searches for sequences in the human genome; IBM WebSphere Business Modeler™ to model and analyze business processes. Finally, and this is the topic of this book, there are simulation modeling tools used to explore various dynamic systems from consumer markets to battlefields.

1.2. Analytical vs. Simulation modeling

If you visit a group responsible for strategic planning, process optimization, sales forecast, logistics, marketing, project management, or HR management in a large company and see what kind of modeling tools and technologies they use you'll find out that the most popular modeling software is MS Excel™. Excel has obvious advantages: it is installed on any office computer and it is very easy to use. It is also extensible: you can add scripts to your formulas as the spreadsheet logic becomes more sophisticated.

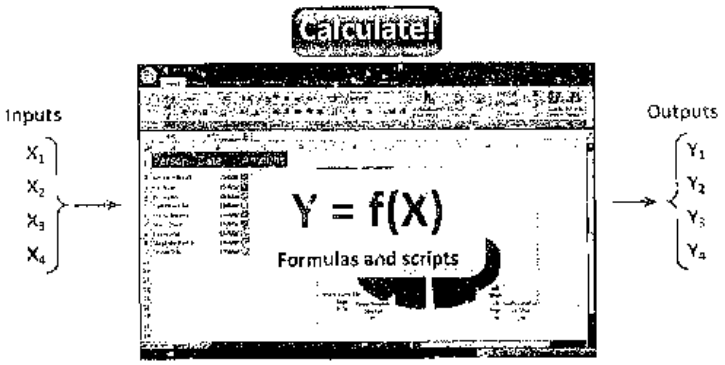


Figure 1.3 Analytical model (Excel spreadsheet)

The technology behind the spreadsheet-based modeling is simple: there are cells where you enter the model inputs and there are other cells where you view the outputs. The output values are linked to the input ones via chains of formulas and, in more complex models, scripts. Various add-ons allow you to perform parameter variation, Monte Carlo, or optimization experiments.

There is, however, a large class of problems where the analytic (formula-based) solution does not exist or is very hard to find. This class, in particular, includes *dynamic systems* featuring:

- Non-linear behavior
- "Memory"
- Non-intuitive influences between variables
- Time and causal dependencies
- All above combined with uncertainty and large number of parameters

You can't even put together a meaningful mental model of such a system not to mention assemble all the appropriate formulas.

Consider as an example a transportation optimization problem where you are to optimize the use of a rail car or truck fleet. Travel, loading and unloading times, maintenances, breakdowns, delivery time restrictions, terminal point capacities make that kind of problem very hard to approach with a spreadsheet. The availability of a vehicle at a particular location on a particular date and time depends on a sequence of events preceding that time. Answering the question of where to send the vehicle when it is idle requires the analysis of event sequences in the future.

- **Formulas that are good for expressing static dependencies between variables, fail to work when it comes to describing the systems with dynamic behavior. This is the time for another modeling technology that is specifically designed for analyzing dynamic systems, namely for *simulation modeling*.**

The *simulation model* is always an *executable model*: you can *run* it and it will build you a trajectory of the systems state changes over time. You can consider a simulation model as a set of rules that tell how to obtain the next state of the system from the current state. Those rules can be of many different forms: differential equations, statecharts, process flowcharts, schedules, etc. The outputs of the model are produced and observed as the model is running.

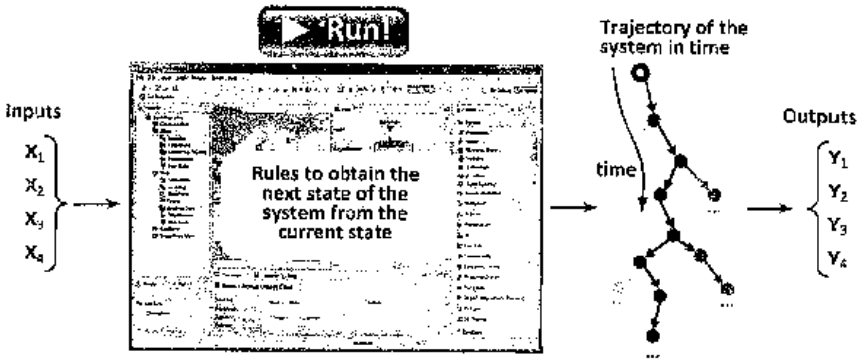


Figure 1.4 Simulation model

Simulation modeling is done with special software tools that employ simulation-specific languages, both graphic and textual. It typically requires some training and learning. But the efforts invested in adoption of simulation technology pay off when you need to perform high quality analysis of a system with dynamic behavior.

People (especially those who count themselves as Excel professionals and have some programming background), nevertheless, sometimes still try to build spreadsheet models of dynamic systems. As they feel the need to capture more details, they inevitably start reproducing the functionality of simulators in Excel. The models become huge and unmanageable. These monsters are full of code, they're slow, they have very short lifetimes, and they are usually soon discarded.

The limits of analytical modeling: queuing theory

To illustrate the power of simulation and to better understand the limits of analytical modeling it is worth spending some time on queuing theory. *Queuing*

theory is a mathematical approach to the analysis of dynamic systems with queues, such as computer transaction processing systems, call centers, transport, customer support, healthcare service systems. Queuing theory was mostly developed in the 1950s and 1960s before the computers became powerful enough to perform (resource-demanding) simulations. It addresses questions like: what is the average number of entities in the queue, what is the distribution of the waiting time, or what is the server utilization.

Consider an example: a bank. On average λ clients per hour enter the bank. At first we will assume there is only one cashier in the bank and on average he serves μ clients per hour (mean service time is $1/\mu$). We are interested in the client's waiting time, queue length and cashier utilization.

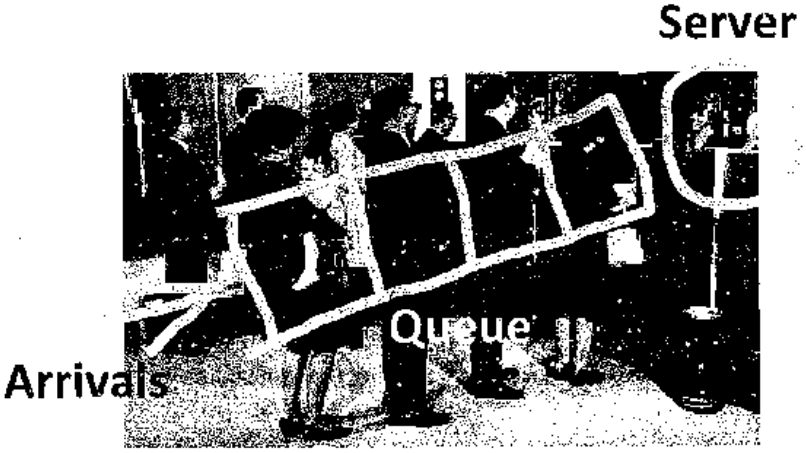


Figure 1.5 A queue in a bank

Queuing theory gives us an easy solution, see case M/M/1 in Figure 1.6. The formulas are very simple and give you the answer immediately. However, the formula for the waiting time is essentially based on two important assumptions:

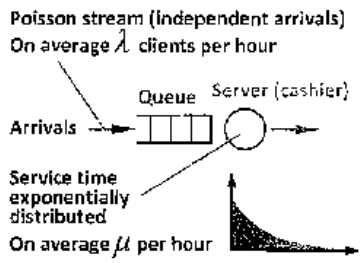
- A *Poisson stream* of clients, and
- Exponentially distributed service time

The first assumption means that the clients arrive at the bank independently, and the time the next client enters the bank door does not depend on the previous client. This looks like a fair assumption for the bank. However, the second assumption does not conform with reality. The distribution of the time spent by a customer at the counter should have some non-zero minimum, a major peak for the most frequent operations, and maybe a second peak for less typical operations (see case M/G/1 in Figure 1.6).The queuing theory does not give up and suggests

another formula for the waiting time that is valid in case of arbitrary distributed service time: Pollaczek–Khinchine formula.

Suppose now that there is not one but three cashiers in the bank. This does not seem to be a big change in the service system. The analytic solution however starts look scary, see case M/M/K. And, moreover, it exists only in the case of exponentially distributed service time. For any other distribution there are no formulas.

And this is it. Any further complication of the bank service process does not have an analytic solution.



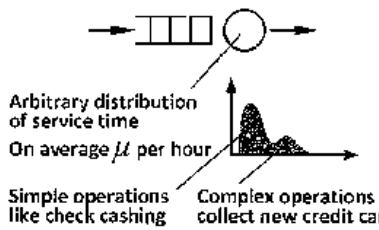
M/M/1

Server utilization*: $\rho = \frac{\lambda}{\mu}$

Average waiting time: $W = \frac{\rho}{\mu - \lambda}$

Average queue length*: $L = \lambda W$ (Little's law)

*These formulas are valid for all cases

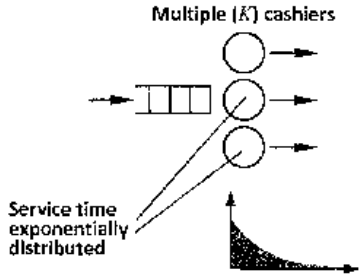

Arbitrary distribution of service time
On average μ per hour
Simple operations like check cashing
Complex operations like collect new credit card

M/G/1

Pollaczek–Khinchine formula:

Average waiting time: $W = \frac{\lambda\left(1 + C_s^2\right)}{2\mu^2(1-\rho)}$

where C_s is coefficient of variation of service time

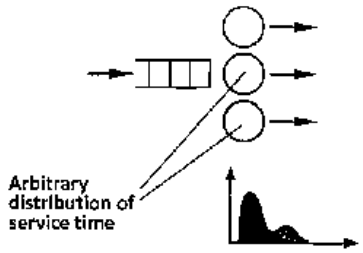


M/M/K

Average waiting time: $W = \frac{P}{K\mu(1-\rho)}$

$$P = \frac{(K\rho)^K}{K!(1-\rho)} P_0$$

$$P_0 = \left[\frac{(K\rho)^K}{K!(1-\rho)} + \sum_{i=0}^{K-1} \frac{(K\rho)^i}{i!} \right]^{-1}$$



M/G/K

Analytic solution does not exist

Any further complication of the service process	Analytic solution does not exist

Figure 1.6 Queuing models of a bank

As you can realize, the process in a real bank is far more complex than even the M/G/K case, for example:

- Some transactions can be done only by some particular employees
- The client can be redirected from one employee to another
- The cashiers may share resources, such as a printer or a copier
- Different cashiers may have different skills and performance
- Etc., etc., etc., ...

Virtually any of those details are impossible to capture in an analytic solution. Even if formulas exist for a particular configuration, a small change in the process may make them void, and you will need a professional mathematician to fix them, most probably from scratch.

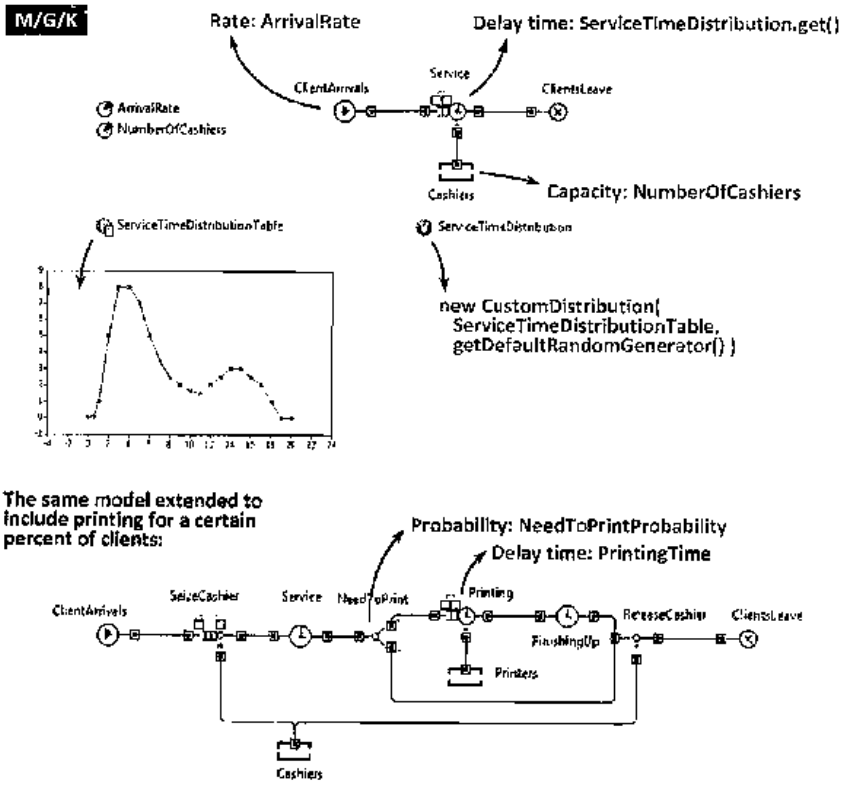


Figure 1.7 Simulation model of a bank

Simulation modeling, on the contrary, can handle service systems of any complexity. Simulation models scale well: adding more details to the service process or making a local change is captured by a corresponding incremental or local change in the simulation model rather than by re-creation of the model from scratch. At the top of Figure 1.7 you can see the simulation model for a bank with an arbitrary number of cashiers, Poisson arrivals and service time with the empirical distribution (M/G/k). At the bottom of Figure 1.7 the model is extended to include printing in a certain percent of cases and sharing a printer between cashiers.

Advantages of simulation modeling

There are six advantages to simulation modeling:

1. Simulation models enable you to analyze systems and find solutions where other methods (like analytic calculations, linear programming, etc.) fail.
2. Once you have selected the appropriate level of abstraction the development of a simulation model is a more straightforward process than analytical modeling. It typically requires less intellectual efforts, is scalable, incremental, and modular.
3. The structure of a simulation model naturally reflects the structure of the real system. As simulation models are developed using mostly visual languages, it is easy to communicate the model internals to other people.
4. In a simulation model you can measure any value and track any entity that is not below the level of abstraction. Measurements and statistical analysis can be added at any time.
5. Ability to play and animate the system behavior in time is one of the greatest advantages of simulation. Animation is used not only for demo purposes, but also for verification and debugging.
6. Simulation models are a lot more convincing than Excel spreadsheets (not to mention Power Point™ slides or reports with numbers). If you bring and run a simulation to support your proposal, you will have an advantage over those who bring just numbers.

1.3. Applications of simulation modeling. Level of abstraction. Methods

Simulation modeling has accumulated a large number of success stories in a very wide and diverse range of applications. And, as new modeling methods and technologies are being developed, and as computer power grows, simulation penetrates new areas.

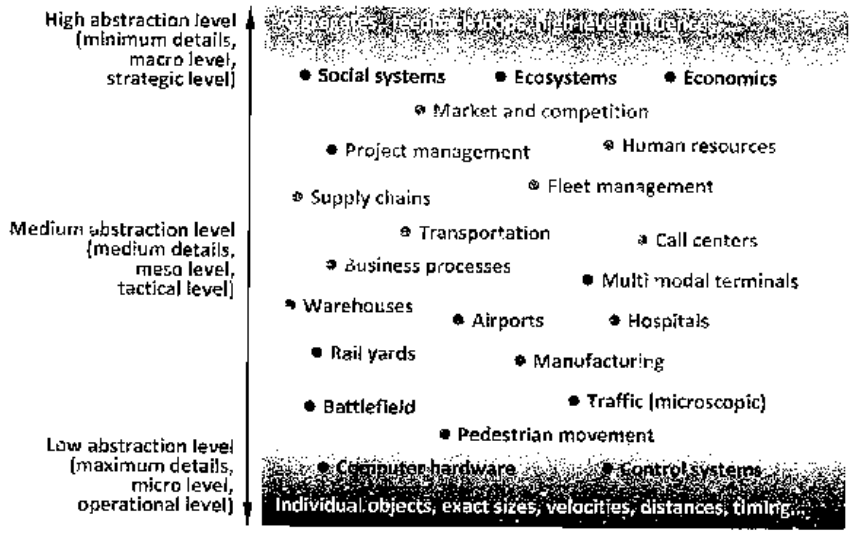


Figure 1.8 Applications of simulation

In Figure 1.8 some applications of simulation are shown sorted by the abstraction level of the corresponding models. The models at the bottom are physical-level models where real world objects are represented with maximum details. At this level we do care about physical interaction, dimensions, velocities, distances, timings. Anti-lock braking system of a car, evacuation of football fans from a stadium, car traffic at an intersection controlled by a traffic light, soldier interaction on a battlefield would be examples of problems that require modeling at a low abstraction level.

The models at the top of the chart are highly abstract. Individual objects are typically replaced there by aggregates. For example, instead of modeling each individual consumer we model the number of consumers, maybe divided into several categories; we model the number of jobs instead of individual jobs, etc. Correspondingly, interaction between the model objects is raised to a high level. In these models the amount of money invested into advertising may directly influence sales, and we do not model the intermediate steps in that causal dependency.

And there are models whose abstraction level is intermediate between low and high. For example, in a model of a hospital emergency department physical space may matter as we do care how long it takes to walk from the emergency care room to x-ray, but physical interaction between people walking in the building is irrelevant because we assume there are no congestions in the building. In a model of a business process or a call center we model the sequence and duration of

operations and do not care about space where those operations take place. In a transportation model we consider trucks or rail car's movement, loading and unloading, whereas in a higher level supply chain model we can assume that shipment of the order takes from 7 to 10 days and we do not care how the shipment is done.

- **Choosing the right abstraction level is critical to the success of the modeling project. Once you have decided what do you include in the model and what is left below the level of abstraction, the choice of the modeling method and the actual "coding" of the model is quite straightforward.**

- **In the model development process it is normal and even desirable to periodically reconsider the abstraction level. Typically you would start with high abstraction and add details as they are needed.**

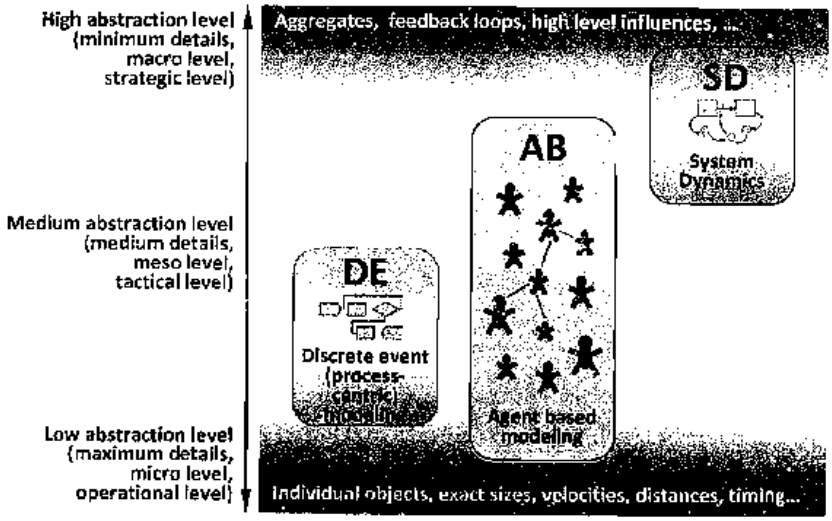


Figure 1.9 Methods in simulation modeling

In modern simulation modeling there are three methods, see Figure 1.9. Each method serves a particular range of abstraction levels. System dynamics operates at high abstraction level and is mostly used for strategic modeling. Discrete event modeling with the underlying process-centric approach supports medium and medium-low abstraction. Agent based models can vary from very detailed where agents modeling physical objects to highly abstract where agent are competing companies or governments. The three methods are considered in detail in Chapter 2.

Chapter 2. The three methods in simulation modeling

By *method* in simulation modeling, we mean a general framework for mapping a real world system to its model. A method suggests a type of language, or "terms and conditions" for model building. To date, there exist three methods:

- System Dynamics
- Discrete Event Modeling
- Agent Based Modeling

The choice of method should be based on the system being modeled and the purpose of the modeling - though often it is most heavily influenced by the background or available tool set of the modeler. Consider Figure 2.1 where the modeler is deciding how best to build a model of a supermarket. Depending on the problem, he may: put together a process flowchart where customers are entities and employees are resources; an agent based model, where consumers are agents affected by ads, communication, and interaction with agents-employees; or a feedback structure, where sales are in the loop with ads, quality of service, pricing, customer loyalty, and other factors.

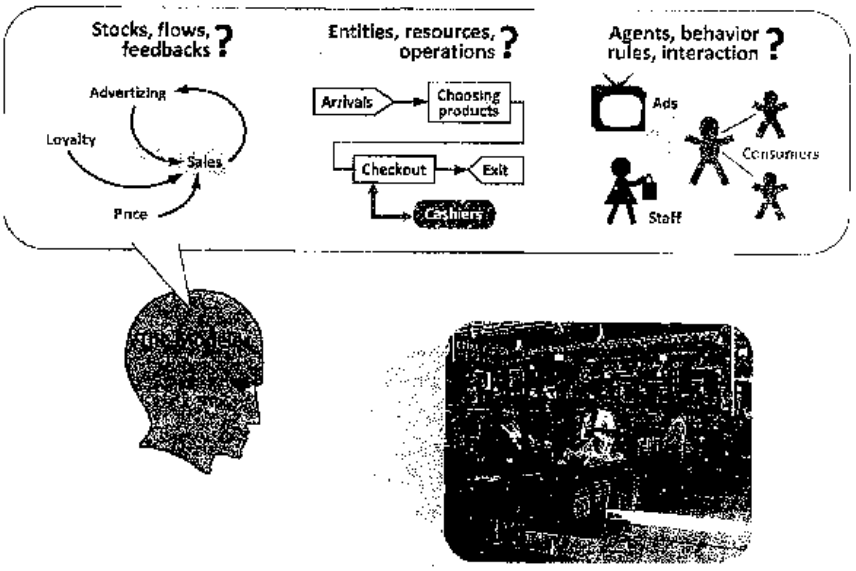


Figure 2.1 The modeler chooses a modeling method

Sometimes, different parts of the system are best (meaning most naturally or elegantly) modeled using different methods. In this case, a multi-method model is built.

2.1. System dynamics

System dynamics is a method created in the mid-1950s by MIT Professor Jay Forrester, whose original background was in science and engineering. Forrester's idea was to use the laws of physics, in particular the laws of electrical circuits, to describe and investigate the dynamics of economic and, later on, social systems. The principles and the modeling language of system dynamics were formed in the 1950s and early 1960s, and remain unchanged today. Most of the definitions below are taken from (System Dynamics Society, Inc., 2013), Wikipedia, and (Sterman, 2000).

- **System dynamics is a method of studying dynamic systems. It suggests that you should:**
  - **Take an endogenous point of view. Model the system as a causally closed structure that itself defines its behavior.**
  - **Discover the feedback loops (circular causality) in the system. Feedback loops are the heart of system dynamics.**
  - **Identify stocks (accumulations) and the flows that affect them. Stocks are the memory of the system, and sources of disequilibrium.**
  - **See things from a certain perspective. Consider individual events and decisions as "surface phenomena that ride on an underlying tide of system structure and behavior." Take a continuous view where events and decisions are blurred.**

To feel the spirit of system dynamics (especially in the context of comparing different modeling methods), consider a shop with a counterman serving the shop's clients. The more people that come to the shop per hour, the longer the queue grows. You can build a discrete event model that will give you the length of the queue as a function of the clients' arrival rate and the service time. However, in a real shop, as the queue grows longer, some clients may decide not to join the queue, and instead leave the shop. Others may decide to leave the queue after having waited longer than they expected to. In other words, the length of the queue feeds back to inhibit the rate of queue growth. The results of the "straightforward" model (sometimes called open-loop models in the system dynamics community), will not be valid unless it addresses these circular causal dependencies. One of the key advantages of the system dynamics approach is to readily and elegantly identify such feedback loops and include them into the model.

This section should not be considered an introduction to system dynamics. This is just a glimpse into a complex area of study and we will provide just one possible scenario of model building to illustrate the methodology. The recommended reading is (Sterman, 2000), one of the best books on modeling in general ever written.

Example 2.1: New product diffusion

Consider a company that starts selling a new consumer product. The addressable market has a known size, which does not change over time. Consumers are sensitive to both advertising and word-of-mouth. The product has an unlimited lifetime and does not need replacement or repeated purchases. A consumer needs only one product. We are to forecast the sales dynamics.

We will start with identifying the key variables in our model, and will iteratively draw *causal loop diagrams*. In a causal loop diagram, variables are connected by arrows showing the causal influences among them, with important feedback loops explicitly identified. Causal loop diagrams are good for quickly capturing your hypotheses about the causes of the dynamics in the system, and communicating it to others (Sterman, 2000).

In our system, one of the variables is obviously **Sales** – the number of people who bought our product per time unit, e.g. per week. The number of **Potential Clients** will be the other variable. The bigger the market, the greater the sales; therefore, we can draw a causal dependency from **PotentialClients** to **Sales** with positive polarity (see Figure 2.2, A). On the other hand, as potential clients buy the product, they stop being potential clients, so there is another influence from **Sales** back to **PotentialClients**, this time with negative polarity. The feedback loop we have just created is a *negative*, or *balancing feedback*, loop: it works for reaching a certain goal. In our case, we ultimately will sell the product to all potential clients, and both variables will become zero.

What determines the sales rate? According to our assumptions, consumers are sensitive to ads and to what other consumers say. So, we will distinguish between sales from advertising, and sales from word of mouth. We introduce two new variables and create two balancing loops instead of one, see Figure 2.2, B. The **SalesFromWordofMouth** depend on the number of (hopefully happy) owners of our product – our **Clients**. The number of clients grows with **Sales**. We draw another feedback loop, this time *positive, or reinforcing* (see Figure 2.2, C).

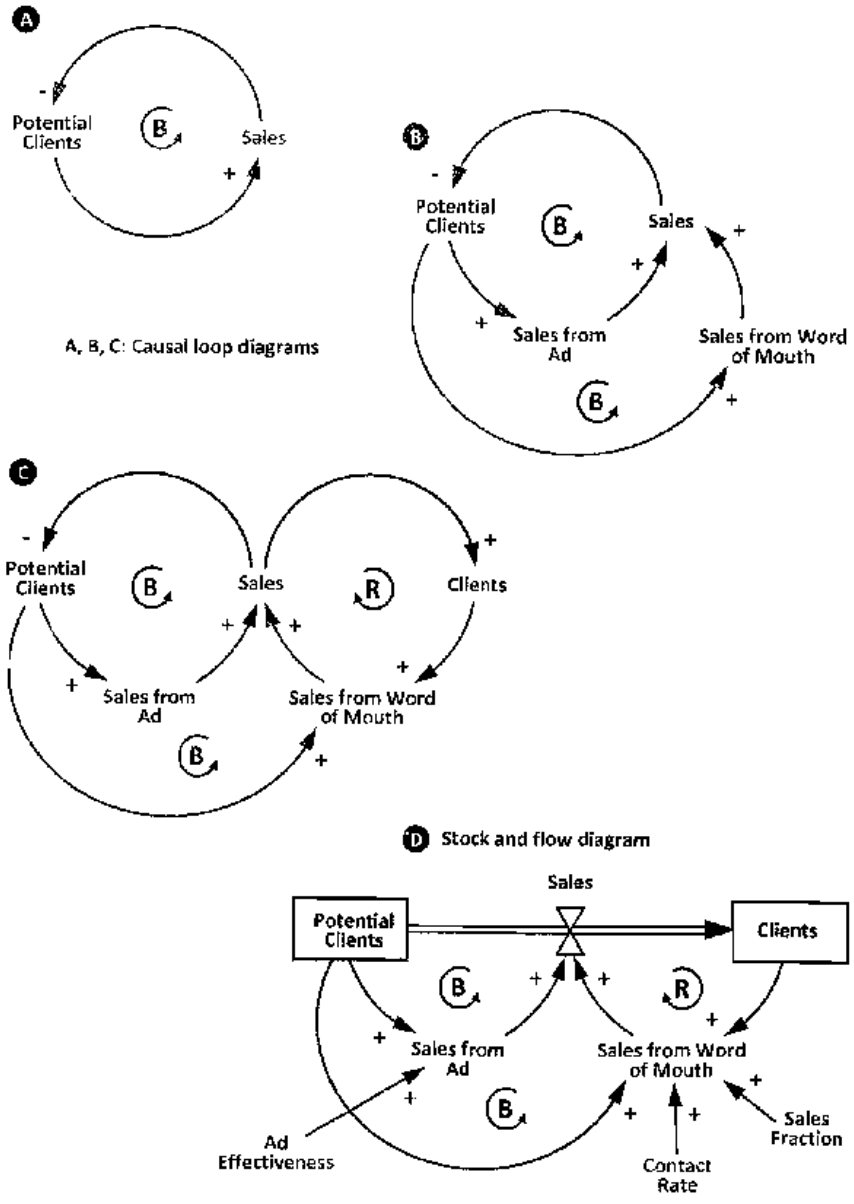


Figure 2.2 System dynamics model of a new product diffusion

While in the causal loop diagram we have drawn shows variable interdependencies and feedbacks, it misses the clear mathematical interpretation, and therefore cannot be simulated directly. One of the things that we need to do on our way to the mathematical model is to identify *stocks and flows* among the variables in our system. Stocks are accumulations, and characterize the state of the system. Flows are the rates at which these system states change. Units of measure can help identify stocks and flows. Stocks are usually quantities such as people, inventory, money, and knowledge. Flows are measured in the same units per time period; for example, clients per month, or dollars per year.

In our model, the stocks are **PotentialClients** and **Clients**, and the flow between them is **Sales**. We can now draw a *stock and flow diagram* and write equations for our model. The diagram is shown in Figure 2.2, D. The equations behind that diagram are:

$$\frac{d(PotentialClients)}{dt} = -Sales$$

$$\frac{d(Clients)}{dt} = Sales$$

$$Sales = SalesFromAd + SalesFromWordofMouth$$

The first two equations are *differential equations*. They define how the stock values change over time. For example, the number of **Clients** grows at the **Sales** rate. The third equation tells that the sales rate consists of two sources, and those sources are independent. The equations for those sources, however, are not clear from the causal loop diagram, and we need to make more assumptions in order to define them.

How big are the sales from advertising? Frank Bass, who originally developed this model, assumed that the probability that a potential client will decide to buy as a result of exposure to advertising is constant at each time period. Therefore, at each period, advertising causes a *constant fraction of potential clients* to buy the product. To reflect this in the model, we introduce a parameter, **AdEffectiveness**, which is a fractional rate. The equation for the sales from advertising, therefore, is:

$$SalesFromAd = PotentialClients \times AdEffectiveness$$

The equation for the sales generated by word of mouth is common for any model of diffusion, be it infectious disease, innovation, a new idea, or a new product. We assume that the people in the community come into contact at a certain rate -- say, each person contacts **ContactRate** people per time unit. If a client contacts a potential client, the latter will decide to buy with a certain constant probability, which we will call **SalesFraction**. So, at each time unit, each client generates

$$ContactRate \times \frac{PotentialClients}{PotentialClients + Clients} \times SalesFraction$$

sales. The middle component in this formula is the fraction of potential clients among all people in the community -- that is, the probability that a person being contacted is a potential client. The formula for the sales from word of mouth, then, is the following:

$$SaleFromWordofMouth =$$

$$Clients \times ContactRate \times \frac{PotentialClients}{PotentialClients + Clients} \times SalesFraction$$

Now the mathematical representation of the model is completed, and we are almost ready to run the simulation.

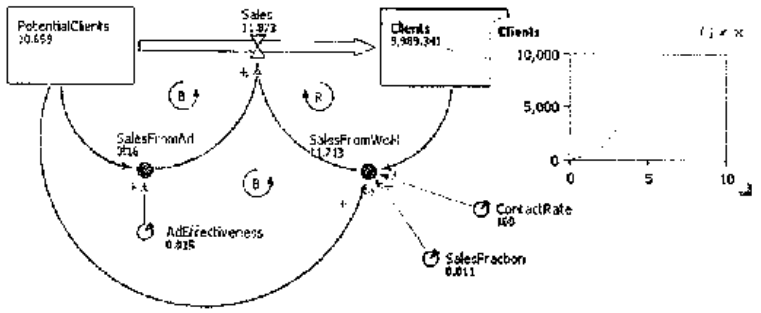
The model-building scenario described above is just one of many possible scenarios. For example, a modeler can start thinking in terms of stocks and flows right away and not use the causal loop diagram at all, or generate it afterwards from the stock and flow diagram. One could start with the positive feedback loop describing the word of mouth effect, and then add the advertising influence. The choice of additional assumptions and the corresponding parameters could also be different.

Before running the simulation, however, we need to specify the values of the parameters and the initial values of the stocks. While the size of the market is roughly known, and the number of clients is known for sure, the three parameters that we have introduced in the last phase of model building cannot be measured directly in the real system. One of the ways to determine the values of such parameters is through *model calibration*. Calibration assumes you have reference (historical) data, with which you can compare and fit the simulation output by varying the model parameters.

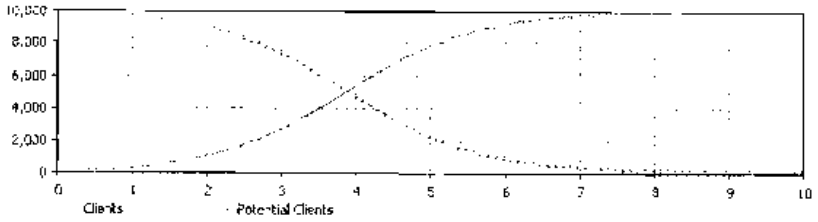
Figure 2.3 shows the simulation results for the following parameter values:

- The size of the market is 10,000 people
- At each time unit, 1.5% of potential clients buy the product because of advertising
- Each person contacts 100 other people per time unit
- If a client contacts a potential client, the latter will buy the product with the probability of 1.1%.

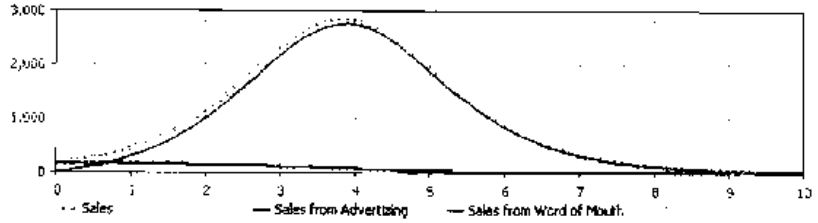
AnyLogic system dynamics model at runtime



S-shaped curve of Client base (= cumulative Sales)



Bell-shaped curve of Sales and its components



Sales curve if advertizing is stopped shortly after the product launch

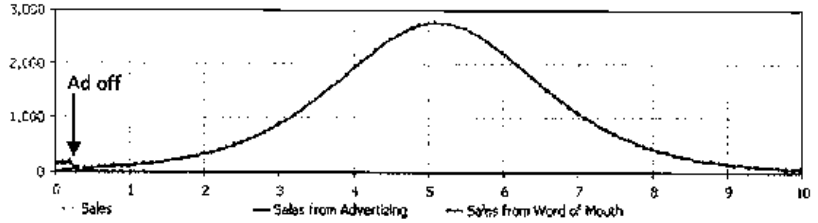


Figure 2.3 Output of the new product diffusion model

The curve of the client base over time is S-shaped. This is a result of the interaction of the two feedback loops. In the beginning, while the size of the market is still large, the reinforcing word-of-mouth effect causes the exponential growth. Then, as the market gets closer to saturation (meaning the system approaches its limit growth limit), the balancing feedback loop starts to dominate and brings the system to the equilibrium state: everybody who could buy the product has now bought it, and sales are at zero .

The sales curve (sales is derivative of the client base) is bell-shaped. You can compare two scenarios; in the one at the very bottom of Figure 2.3, the advertising is turned off shortly after the launch of the product, which has a surprisingly small effect on sales.

Underlying mathematics and simulation engine

Mathematically, a system dynamics model is a system of coupled, nonlinear, first-order differential equations

$$\frac{d(X)}{dt} = F(X, P)$$

where X is a vector of stocks, P is a set of parameters, and F is a nonlinear vector-valued function. Simulation of system dynamics models is done with *numerical methods* that partition simulated time into discrete intervals of length dt and step the system through time one dt at a time.

While numerical methods may be very sophisticated in the modeling tools used by natural scientists and engineers, in particular using adaptive variable time step, the numerical methods used in system dynamics are simple, fixed-step methods: Euler and Runge-Kutta. In addition to differential equations, the simulation engine must be able to solve algebraic equations that appear in the models with *algebraic loops*.

Unlike discrete event and agent-based models, system dynamics models are deterministic, unless stochastic elements are explicitly inserted into them.

Abstraction level

System dynamics suggests a very high abstraction level, and is positioned as a strategic modeling methodology. In the models of social dynamics, epidemics, or consumer choice, individual people never appear as well as individual product items, jobs, or houses – they are aggregated into stocks (compartments) and sometimes segmented into gender, education, income level, etc. (You have probably noticed non-integer values such as 10.659 people or 0.24 cars in system dynamics models during runtime.) Similarly, individual events like a purchase decision,

leaving a job, or recovery from a disease, are not considered – they are aggregated in flows.

Although the language of system dynamics is very simple, if not primitive, compared to other methods, thinking in its terms and on its level of abstraction is a real art. System dynamics models are inevitably full of notions that do not have direct material or measurable equivalents in the real world -- for example, morale, awareness, knowledge, and the impact of advertising. The choice of those notions, and drawing the corresponding feedback structures, is not, in many cases, as straightforward a task as the design of a process flowchart or agent behavior.

Software tools

System dynamics modeling is supported by four software tools: Vensim™, AnyLogic™, iThink™/STELLA™, and Powersim™. The modeling language of system dynamics is well-defined and minimalistic -- there are no application-specific dialects. Model conversion is possible between most pairs of tools, and vendors are discussing using a standard XML-based language to be supported by everyone. The tools differ in details. For example, iThink offers special types of stocks like Queue or Oven, which are not present in other tools; Vensim and AnyLogic offer more powerful and flexible arrays; etc.

2.2. Discrete event modeling

Discrete event modeling is almost as old as system dynamics. In October 1961, IBM engineer Geoffrey Gordon introduced the first version of GPSS (General Purpose Simulation System, originally Gordon's Programmable Simulation System), which is considered to be the first method of software implementation of discrete event modeling. These days, discrete event modeling is supported by a large number of software tools, including modern versions of GPSS itself.

- **The idea of discrete event modeling method is this: the modeler considers the system being modeled as a process, i.e. a sequence of operations being performed across entities.**

The operations include delays, service by various resources, choosing the process branch, splitting, combining, and some others. As long as entities compete for resources and can be delayed, queues are present in virtually any discrete event model. The model is specified graphically as a process flowchart, where blocks represent operations (there are textual languages as well, but they are in the minority). The flowchart usually begins with "source" blocks that generate entities and inject them into the process, and ends with "sink" blocks that remove entities

from the model. This type of diagram is familiar to the business world as a process diagram and is ubiquitous in describing their process steps. This familiarity is one of the reasons why discrete event modeling has been the most successful method in penetrating the business community.

Entities (originally in GPSS they were called *transactions*) may represent clients, patients, phone calls, documents (physical and electronic), parts, products, pallets, computer transactions, vehicles, tasks, projects, and ideas. *Resources* represent various staff, doctors, operators, workers, servers, CPUs, computer memory, equipment, and transport.

Service times, as well as entity arrival times, are usually stochastic, drawn from a probability distribution. Therefore, discrete event models are stochastic themselves. This means that a model must be run for a certain time, and/or needs a certain number of replications, before it produces a meaningful output.

The typical output expected from a discrete event model is:

- Utilization of resources
- Time spent in the system or its part by an entity
- Waiting times
- Queue lengths
- System throughput
- Bottlenecks
- Cost of the entity processing and its structure

Example 2.2: Bank

Consider a bank with an ATM inside. The process in the bank is described as follows:

- On average, 45 clients per hour enter the bank.
- Having entered the bank, half of the clients go to the ATM, and the other half go straight to the cashiers.
- Usage of the ATM has a minimum duration of 1 minute, a maximum of 4 minutes, and a most-likely duration of 2 minutes.
- Service with a cashier takes a minimum of 3 minutes and a maximum of 20 minutes, with a most-likely duration of 5 minutes.
- After using the ATM, 30% of the clients go to the cashiers. The others exit the bank.
- There are 5 cashiers in the bank, and there is a single shared queue for all the cashiers.
- After being served by a cashier, clients exit the bank.

We need to find out the:

- Utilization of cashiers
- Average queue lengths, both to the ATM and to the cashiers, and the
- Distribution of time spent by a customer in the bank.

With this problem definition, building a discrete event model is more or less a straightforward task. Clients obviously are entities, and the cashiers are resources. The flowchart of the bank is show in Figure 2.4. The block **ClientsArrive** generates clients at the rate 0.75 per minute (45 per hour). Having appeared in the model, 50% of the clients go to the cashiers, and 50% to the ATM. The usage of the ATM is modeled by a **Delay** block **ServiceAtATM**, preceded by a **Queue** block. Service at cashiers is modeled by a pair **Service** with triangularly distributed service time and **ResourcePool Cashiers** with capacity 5. The flowchart ends with the **Sink** block **ClientsLeave**.

Despite the example above is not always easy to identify what are entities and what are resources, especially in systems where there is no obvious continuous process flow (many manufacturing systems fall into this class). Sometimes you need to introduce a "virtual" entity to establish causality and the sequence of operation. If this is so, it may make sense to consider the agent-based modeling method and represent the system as a set of interacting active units. This is discussed later in this chapter.

The output data is generated as the model is running. Statistics are collected at the blocks, as well as by the entities as they move through the process flowchart. For example, the cashiers' utilization is a part of the standard report offered by the **Resource Pool** object. Each entity (client) measures time spent in the bank by making a timestamp at the entry and then comparing it with the current time at the exit.

Abstraction level

As you can see, the level of abstraction suggested by discrete event modeling is significantly lower than that of system dynamics; the diagram mirrors sequential steps that happen in the physical system. While in system dynamics we aggregate individual objects and talk about the dynamics of their quantities, in discrete event modeling each object in the system is represented by an entity or a resource unit, and keeps its individuality. Entities and resources may have attributes, may differ from each other, and can be treated differently by the process (the "perfect mix" assumption of a system dynamics model is dropped). In discrete event models, time delays can be deterministic or stochastic with any probability distribution. In system dynamics, "natural" delays have an exponential distribution and deterministic delays are special constructs.

Another important fact about process models—and this is the main difference with the agent based models we'll discuss next —is that both entities and resource units are *passive*: they have no behavior "on their own", they just carry their data. Anything that happens to them is defined by the process flowchart.

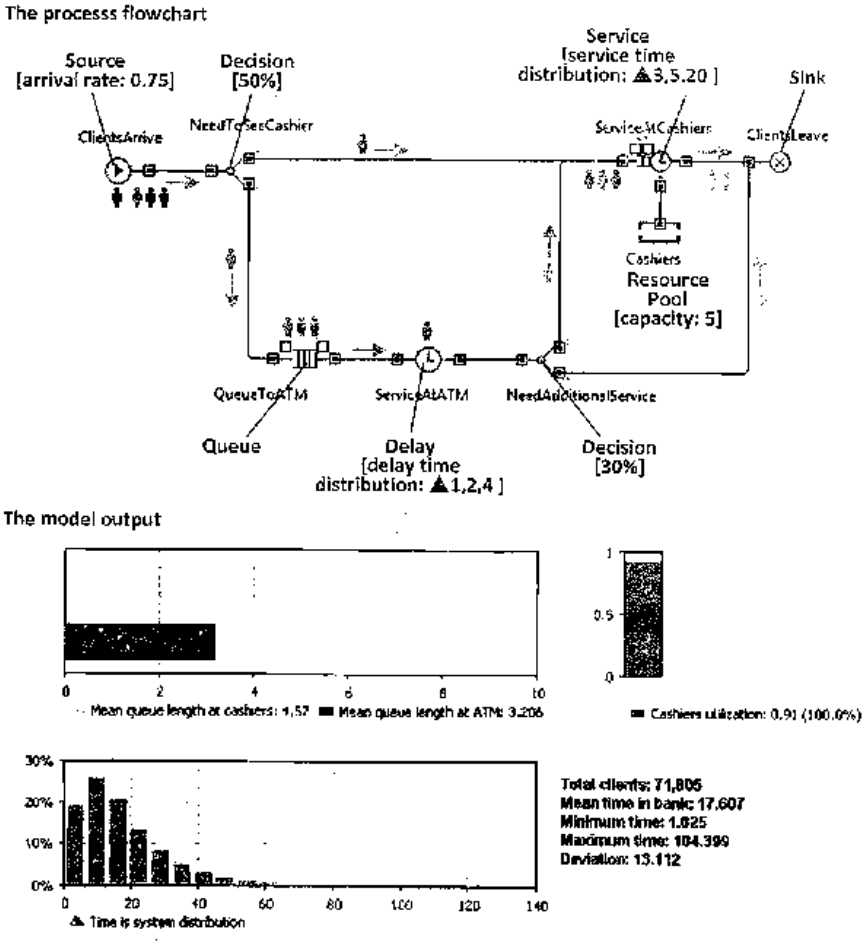


Figure 2.4 Discrete event model of a bank

Underlying mathematics and simulation engine

The mathematics behind discrete event simulation are based on *discrete time*. The model clock is advanced only when something significant happens in the model -- namely, when an entity starts or finishes an operation. Any change in the model is

associated with those events; continuous changes are approximated by instantaneous ones. Discrete time, and its implementation in the simulation engine, is discussed in more details in Chapter 8, "Discrete events and Event model onject".

Software tools

Unlike system dynamics, discrete event modeling is supported by tens (if not hundreds) of software tools. There is no uniformly accepted language for specifying discrete event models; interoperability is not possible, and not even planned by software vendors. There are some standardized languages for specifying subclasses of processes -- for example, business processes -- but even those languages are not directly supported by simulation modeling tools. Each tool offers its own set of blocks and its own scripting language. Some tools are general-purpose, while some are specifically designed for a particular application area. They may look very different, but behind the GUI each one of them has a simulation engine that supports a discrete clock and event queue, and moves entities through the process flowchart.

2.3. Agent based modeling

Agent based modeling is a more recent modeling method than system dynamics or discrete event modeling. Until the early 2000s, agent based modeling was pretty much an academic topic. The adoption of agent based modeling by simulation practitioners started in 2002-2003. It was triggered by:

- Desire to get a deeper insight into systems that are not well-captured by traditional modeling approaches
- Advances in modeling technology coming from computer science, namely object oriented modeling, UML, and statecharts (see Chapter 7)
- Rapid growth of the availability of CPU power and memory (agent based models are more demanding of both, compared to system dynamics and discrete event models)

Agent based modeling suggests to the modeler yet another way of looking at the system.

- **You may not know how the system as a whole behaves, what are the key variables and dependencies between them, or simply don't see that there is a process flow, but you may have some insight into how the objects in the system behave individually. Therefore, you can start building the model from the bottom up by identifying those objects (agents) and defining their behaviors.**

- **Sometimes, you can connect the agents to each other and let them interact; other times, you can put them in an environment, which may have its own dynamics. The global behavior of the system then emerges out of many (tens, hundreds, thousands, even millions) concurrent individual behaviors.**

There are no standard languages for agent based modeling. The structure of an agent based model is created using graphical editors or scripts, depending on the software. The behavior of agents is specified in many different ways. Frequently, the agent has a notion of state, and its actions and reactions depend on its state. In such cases, behavior is best defined with statecharts. Sometimes, behavior is defined in the form of rules executed upon special events. In many cases, the internal dynamics of the agent can be best captured using system dynamics or discrete event approach. In these cases, we can put a stock and flow diagram (see Section 5.1) or a process flowchart inside an agent. Similarly, outside agents and the dynamics of the environment where they live are often naturally modeled using traditional methods. We find that a large percentage agent based models, therefore, are multi-method models.

Example 2.3: Agent based epidemic model

As an example we will build an agent based model of the spread of contagious disease. Here is the problem statement:

- Consider a population of 10,000 people. They live in an area measuring 10 by 10 kilometers, and are evenly spread throughout the area.
- A person in the area knows everybody who lives within 1 kilometer of him, and does not know anybody else.
- 10 random people are initially infected, and everybody else is susceptible (none are immune).
- If an infectious person contacts a susceptible person, the latter gets infected with probability 0.1.
- Having been infected, a person does not immediately become infectious. There is a latent phase that lasts from 3 to 6 days. We will call people in the latent phase *exposed*.
- The illness duration after the latent phase (i.e. the duration of the infectious phase) is uniformly distributed between 7 and 15 days.
- During the infectious phase, a person on average contacts 5 people he knows per day.
- When the person recovers, he becomes immune to the disease, but not forever. Immunity lasts from 30 to 70 days.

We are to find out the epidemic dynamics -- namely, the number of exposed and infectious people over time.

The terminology and the overall structure of the problem is taken from the ("Compartmental models in epidemiology", n.d.) -- namely, from the SEIR (Susceptible Exposed Infectious Recovered) model. The SEIR problem was originally solved using differential equations; the approach is similar to system dynamics. We, however, are adding details that are not well captured by compartmental (aggregated) models: space, communication dependent on space, and uniformly distributed phase durations. The rationale behind using the agent based approach is its *naturalness*: we may not know how to derive global equations for a particular disease, but we know the course of the disease and can model this easily and in a straightforward manner at the individual level.

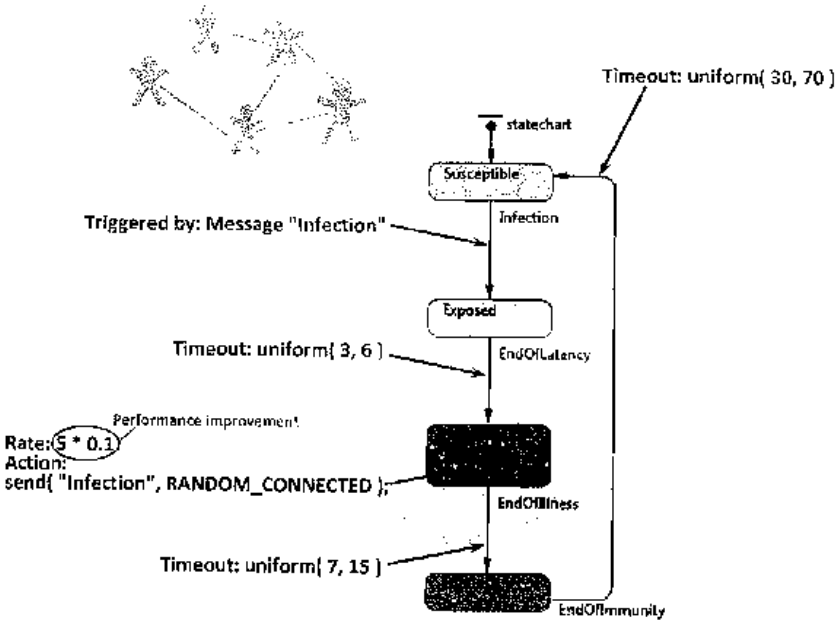


Figure 2.5 Individual behavior of a person in the agent based epidemic model

According to the problem specification, a person can potentially be in one of four phases, and different things happen to him in each of the different phases. This type of behavior is best modeled with a statechart (see Chapter 7). Look at Figure 2.5. The four states of the statechart correspond to the four phases of the disease. Initially, the person is in the `Susceptible` state, and will remain susceptible unless gets infected. Infection can only be passed during a contact; it is modeled by a message

sent from one agent to another. The transition from **Susceptible** to **Exposed** is triggered by the receipt of the message "**Infection**". Once the person gets to the **Exposed** state, the timeout for the next transition **EndOfLatency** is evaluated, and the transition gets scheduled. Thus, the time spent in the **Exposed** state is drawn from the uniform distribution with the range [3, 6] days. Similarly, the time spent in the **Infectious** and **Immune** states is defined by the corresponding stochastic timeouts.

Finally, while the person is in the **Infectious** phase, he periodically contacts other people and can pass infection. This is modeled by an internal transition (see Section 7.3) **Contact** that is executed cyclically at a given rate. When that transition is taken, a message "**Infection**" is sent to a randomly chosen "friend".

Here we do a simple simulation performance improvement. Instead of explicitly modeling each contact and then deciding whether the other person was infected, we model only "successful" contacts, which are far less common. We multiply the contact rate by the infection probability, so that if the latter is 10%, the contacts will occur 10 times more rarely, but will pass infection for sure.

Now that we have defined the behavior of a person, we need to:

- Evenly populate the 2D space with 10,000 people (in object-oriented terminology, we would say *instances* of class **Person**),
- Establish network connections based on distance (this type of network is supported by most agent based modeling tools)
- Infect 10 random people (e.g. by manually sending them the message "**Infection**" at the beginning of the simulation)
- Run the model.

The dynamics of the model are fascinating to watch. Look at Figure 2.6. Shortly after the model initializes, you can observe the clusters of sick people around the initially infected ones. After the initial peak at about day 50, the epidemic goes down. However, because the immunity of the recovered people does not last very long, they get infected again by those people who are still sick. The second wave of the epidemic starts, and it all repeats again.

It is exciting to play with the parameters. If we run the same model with a longer and more deterministic immunity phase, say from 60 to 70 days, we will observe only one outbreak of the epidemic and no oscillations (see the bottom chart in Figure 2.6). If we then change the type of network to one with some long distance links, the epidemic will again last forever.

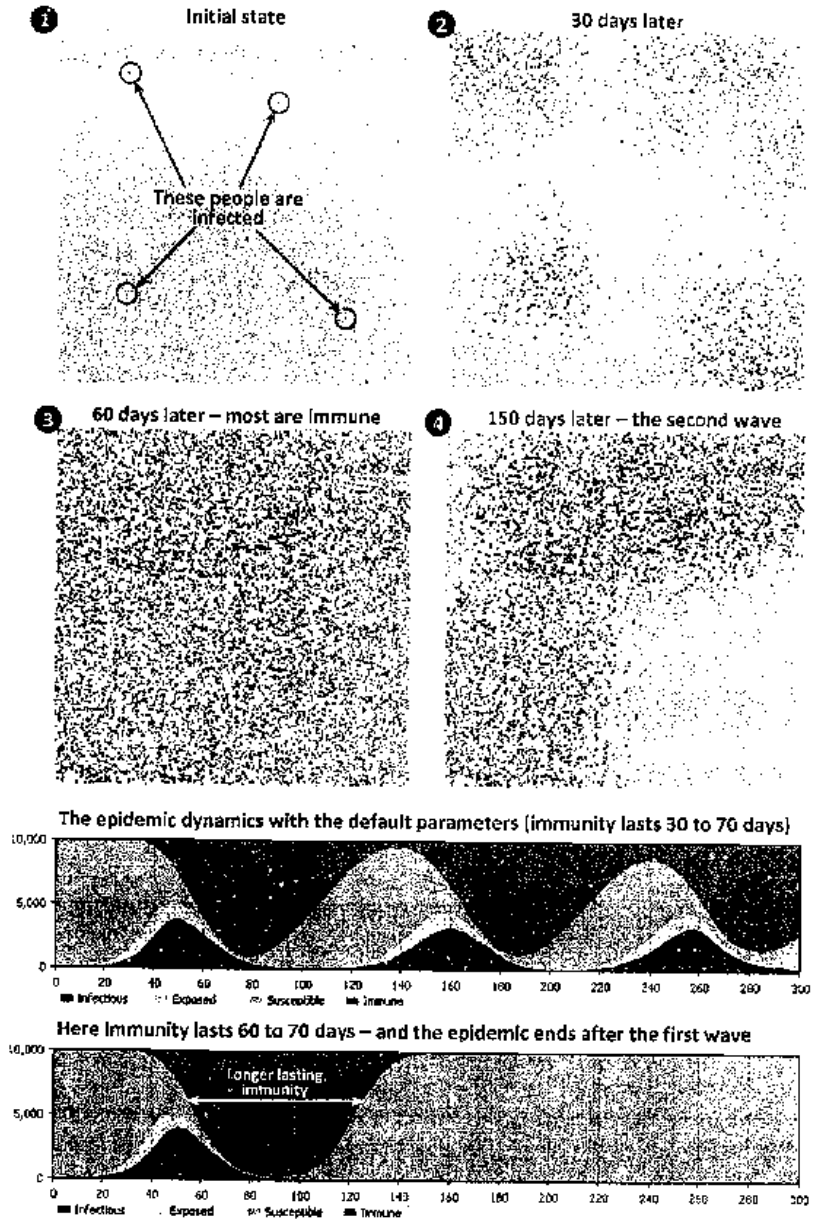


Figure 2.6 Animation and output of the agent based epidemic model

The agent based model we have created is very easy to modify to represent new assumptions. Different types of social networks, households, contacts at work and at home, treatment, impact of vaccination, etc. – all can be captured in the agent based model, naturally and incrementally.

Abstraction level

Agent based modeling does not assume any particular abstraction level. If agents are individuals, then the agent based model is certainly more detailed than a segmented system dynamics model where individuals are aggregated based on characteristics. Agent, however, can be developed with high level of abstraction. For example the agents may be competing projects, companies, or even ideas or philosophies.

Deciding what, exactly, should be modeled as an agent is not always as trivial as in the epidemic example. Even if we are considering people, an individual person should not necessarily become an agent. For example, in the model of an automobile market, agents may be households and not individual people, because the decision about which car to buy is mainly made at the household level and depends on household parameters.

For those who have read books and papers on agent based modeling

Academics are still discussing what kind of properties an object should have to "deserve" to be called an "agent": pro- and re-activeness, spatial awareness, ability to learn, social ability, "intellect", etc. In applied agent based modeling, however, you will find all kinds of agents: stupid and smart, communicating with each other and living in total isolation, living in space and without a space, learning and adapting and as well as not changing their behavior patterns at all, and so on.

Here are some useful facts aimed to ensure that you are not misguided by academic literature and various "theories" of agent based modeling:

- **Agents are not the same thing as cellular automata** and they do not have to live in discrete space (like the grid in *The Game of Life*, ("The Game of Life", n.d.]). In many agent based models, space is not present at all. When space is needed, in most cases it is continuous -- sometimes a geographical map or a facility floor plan.
- **Agent based modeling does not assume clock "ticks"** or "steps" on which agents test conditions and make decisions (*synchronous discrete time*). Most well-built and efficient agent based models are *asynchronous* (conditions are tested and things happen only when they need to). Continuous time dynamics may also be a part of agent or environment behavior.

- **Agents are not necessarily people.** Anything can be an agent: a vehicle, a piece of equipment, a project, an idea, an organization, an investment, etc. For example, a model of a steel converter plant where each machine is modeled as an active object and steel is produced as a result of their interaction is an agent based model.
- **An object that seems to be absolutely passive can be an agent.** For example, even a pipe segment in a water supply network can be modeled as agent. We can associate it with maintenance and replacement schedules, cost, breakdown events, and so on.
- **There can be many, or very few, agents in an agent based model** (compare the model of the American automobile market and the model of the steel converter plant). Agents can be of the same type, or can be all of different types.
- **There are agent based models where agents do not interact at all.** For example, in the area of health economics, there are models of alcohol usage, obesity, chronic diseases where individual dynamics depend only on personal parameters, and, sometimes, on the environment.

Underlying mathematics and simulation engine

Most agent based models work in discrete time -- interaction, decision making, and state changes are instant. In this respect, at the low level the simulation engine should not be much different from the one used for discrete event modeling. At a higher level, it is desirable that the engine supports:

- A large number of concurrent activities, including their dynamic creation and destruction.
- Correct handling of multiple instantaneous events, in particular deterministic and random execution. This is important for synchronous models.
- Networks and communication.
- 2D, 3D, and geographical space, and space-related functionality.

If the agent internal dynamics or environment dynamics have continuous-time elements, such as differential equations, the simulation engine must include numerical methods and support hybrid discrete/continuous time.

Software tools

AnyLogic remains the only professional agent based modeling tool. It supports statecharts (see Chapter 7) and action charts, object oriented-ness and Java, and has the ability to use system dynamics and process flowcharts inside and outside agents to allow the building of industrial strength agent based models. Other software for agent based modeling falls in one of the two classes: graphical tools with limited

capabilities for rapid building of toy models, or Java or C++ libraries where you can do more, but have to code a lot.

Chapter 3. Agent based modeling. Technology overview

In this chapter, we will present an overview of the technologies and techniques used in agent based modeling. It is important to understand that there is no special or standard language for agent based modeling. Agent based models are very diverse in architecture, behavior types, number of agents, space, and so on. Other modeling methods (discrete event and system dynamics) are often used inside and outside agents. There are, however, "design patterns" that are common to many agent based models, which we will consider:

- "Object-based" architecture
- Time model: asynchronous or synchronous (steps or clock ticks)
- Space (continuous, discrete, geographical) and mobility
- Networks and links between agents
- Communication between agents, and between agents and environment
- Dynamic creation and destruction of agents
- Statistics collection on agent populations

Before we dive into the technical stuff, we will discuss what kind of real world objects the agents actually represent.

3.1. Who are the agents?

Agents in an agent based model may represent very diverse things: vehicles, units of equipment, projects, products, ideas, organizations, investments, pieces of land, people in different roles, etc.

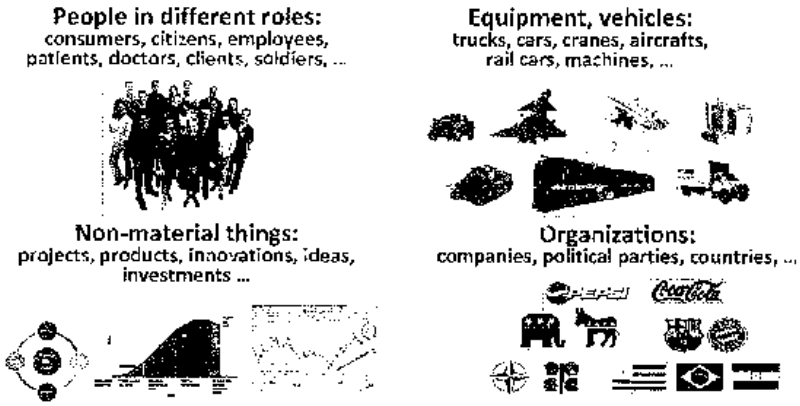


Figure 3.1 Agents may be...

Given a particular problem we are solving with the help of agent based modeling, how do we choose who the agents are? This may not be as trivial as it seems. Consider the following example.

Who are the agents in an American automotive market model?

You are going to model the American automotive market (for example, to forecast its reaction to a certain move by one of the players) and you have decided to use agent based modeling. In automotive market, people buy and sell cars. So, the first thing that comes to mind is that agents are people because they make decisions.

Figure 3.2 Agents are people

What about cars? Cars, at first glance, are passive objects that have different parameters and may vary in appeal to different people. But cars do have some dynamics because with age they lose appeal and value. So, should we model cars as agents or just leave them as plain objects, or entities, being handled by agents/people?

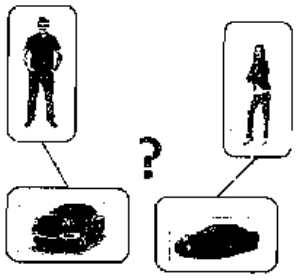


Figure 3.3 Agents are people and cars

What if a person shares a car with a spouse? Who makes the decision then? Should we model collaboration between family members?

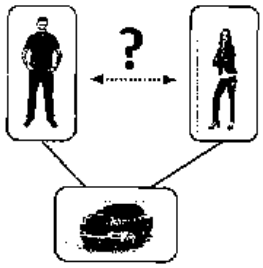


Figure 3.4 A car shared by two family members. Collaborative decision making

The situation gets even more complicated if we consider families with children, families with multiple cars shared by multiple people, and so on.

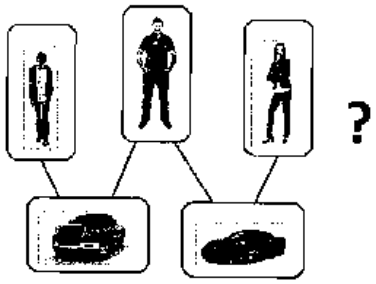


Figure 3.5 Multiple cars shared by multiple people

There is no universal answer to the question. Depending on the details of the problem, different model architectures may make sense. In the Vehicle Market Model created by Mark Paich (Decisio Consulting and Lexidyne), the agents were not individual people, but *households*. Households had garage slots with cars, and cars were not agents, they were objects with properties like year, make, and model.

Decisions about selling or buying a car were made at the household level, taking into account the number of family members, their ages, income, and other factors.

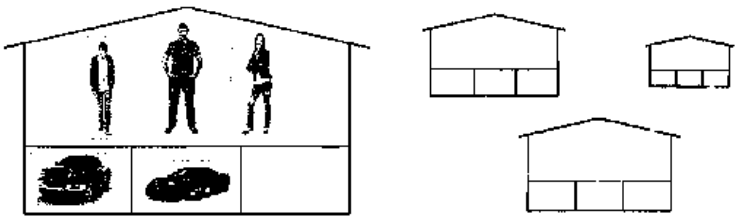


Figure 3.6 Agents are households

3.2. Agent based modeling and object-oriented design

Agent based simulation modeling and object-oriented software design have a lot in common. When software engineers think about how to choose a set of classes, where to draw the line between interface and implementation, and which set of constructors a class should have, they are doing a type of agent based, or individual based, modeling for the problem that the software is addressing.

We can extend this analogy to the *procedural programming paradigm* that historically preceded the object-oriented paradigm. In procedural programming, the software developer was thinking of the program as a (possibly hierarchical) sequence of steps that should be taken to reach a goal. Similarly, in discrete event (process-centric) modeling, the modeler represents the system as a (possibly hierarchical) sequence of operations that are performed over entities. However, unlike procedural programming, which was almost totally replaced by object-oriented programming, discrete event modeling successfully coexists with agent based modeling.

Let's also discuss classes and objects. *Class* is, in a way, a *design-time term*; it is a *description*. Class describes a set of similar objects, but these objects (called *instances* of the class) will not be created until the program is run. Objects exist at runtime. So, speaking rigorously, agents exist only in a running agent based model. When you are developing the model, you only construct *agent classes*.

Agents (objects) of the same class have common structure and behavior, but may differ in details, such as parameter values and *state information* (memory) that includes variable values, statechart and event states, etc. For example, two agents of the class **Patient** have the same set of parameters and the same behavior pattern, but may differ in age, location, disease state, or contact lists.

Interface of the agent (object) is the set of things other agents and external parts of the model can see and use to interact with the agent. These can be variables (in OO terminology these would be "public fields"), functions ("public methods"), ports, or messages. As opposed to interface, *implementation of the agent* (object) is a set of things internal to the agent and hidden from the external world (variables, functions, statecharts, events, embedded agents, etc).

Separation of interface and implementation in OO programming allows for modular development: one can change (optimize, extend, or improve) a class by changing its implementation and other classes will not need to know about the changes as long as the interface stays the same. In OO programming, this separation is strongly supported by the restrictive language constructs that prevent the programmer from accessing the implementation from outside the class (for example, the Java keywords "private" or "protected").

In agent based modeling, those formal access restrictions are often omitted for the purpose of simplicity and more rapid development. For example, the statechart states and transitions in AnyLogic are visible from outside the agent so that one can make inquiries and collect statistics on the agent state by simply writing ***<agent name>.<state name>***. The modeler, however, should not abuse the transparency of the agent border (which he should define himself and keep in mind) and refrain from interacting with the agent in an "illegal" manner. Examples of this would be assigning a new value to the agent variable or resetting the agent internal event from outside.

Important differences between agent based modeling and OO programming are that agents are typically dynamic, have internal time delays, can initiate events, and can have continuously changing variables inside, whereas objects in an OO program typically "act upon request"; they do something only when their methods are called. Agents in a running agent based model, in this sense, are more like *threads* or *concurrent processes* in a running program. This makes the techniques applied in the design of concurrent and parallel systems (such as message sequence diagrams explained in the "Field Service" example, see Section 4.2) useful in agent based modeling.

Similar to objects, agents can be created and destroyed dynamically.

There are things that are fundamentally important in OO design, but are not widely used in agent based modeling. These are, for example, inheritance and polymorphism. Agent based modelers, in general, are not really keen on creating class hierarchies and interfaces or overriding methods in Java or C++ style. Instead, they often use conditional functions and behavior components.

Consider a population model where males and females have differences in behavior. An OO purist would create a class hierarchy like the one shown in Figure 3.7, on the left. Properties and behavior common for all people would be located in the base class **Person**, whereas behavior specific to males and females would be placed in the two corresponding subclasses **Male** and **Female**. A non OO-minded modeler would create just one class **Person**, with the parameter **Sex**, and one behavior component with two branches as shown in the same figure on the right. Both versions may make sense, but which one is better depends on how these constructs are used throughout the model.

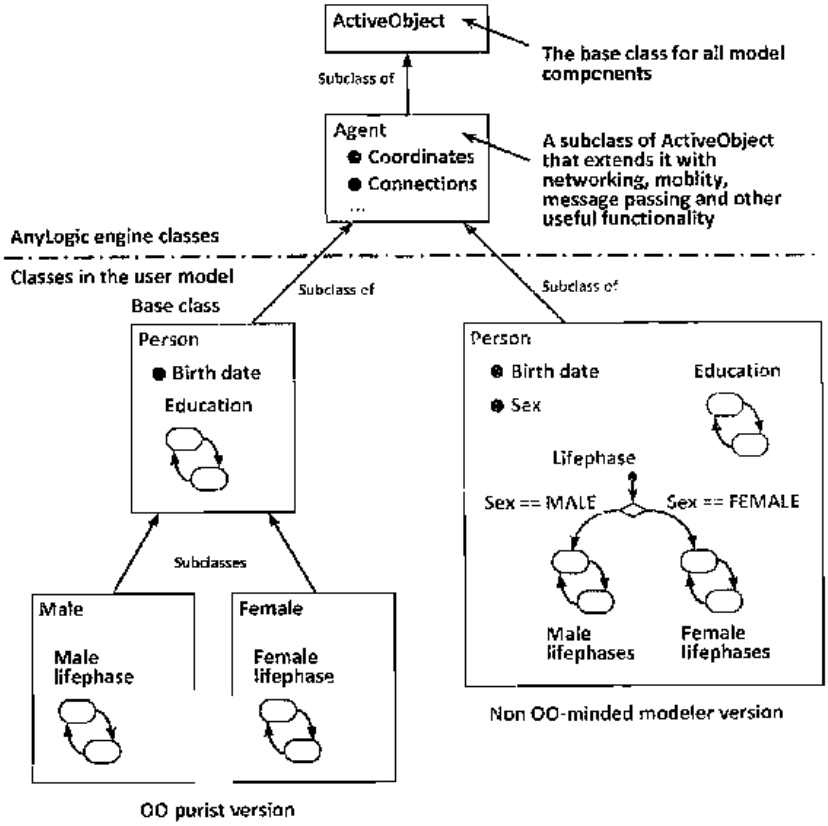


Figure 3.7 OO purist and non OO-minded modeler versions of agent classes in a population model

OO modeling in AnyLogic

Any AnyLogic model, not just an agent based one, consists of classes that inherit from **ActiveObject**. **ActiveObject** is the base class for all model components (called *active objects*) defined in the AnyLogic engine.

In the simplest case, there is just one active object class **Main** in the model. When you create a new model, this class is added automatically and its editor opens. Then you can start to fill the class with process flowcharts, stocks, flows, statecharts, events, and graphics.

In a hierarchical model there is more than one class, and when the model runs objects of different classes are *embedded* one into another. When we say an active object is embedded into another active object (called *container object*), it means:

- The embedded object does not exist without the container; it is created when the container is created, and deleted when the container is deleted.
- If the embedded object is replicated, i.e., has multiple instances, the container object controls dynamic creation and deletion of the instances.
- The animation of the embedded object becomes a part of the animation of the container object ("embedded animation").
- From a Java viewpoint, the embedded object is a member, or field, of the container object and exists in its "namespace" at the same level as variables, parameters, statecharts, etc.

Any agent based model is hierarchical and has at least two classes: the top-level class (like **Main**) that contains the agents, and the agent class (like **Person**). Agents typically exist as a *replicated object* (a collection of multiple objects of the same type) embedded into **Main**, see Figure 3.8. User agent classes do not inherit directly from the **ActiveObject** class; they are subclasses of the class **Agent**, which extends **ActiveObject** with features specific for agent based modeling.

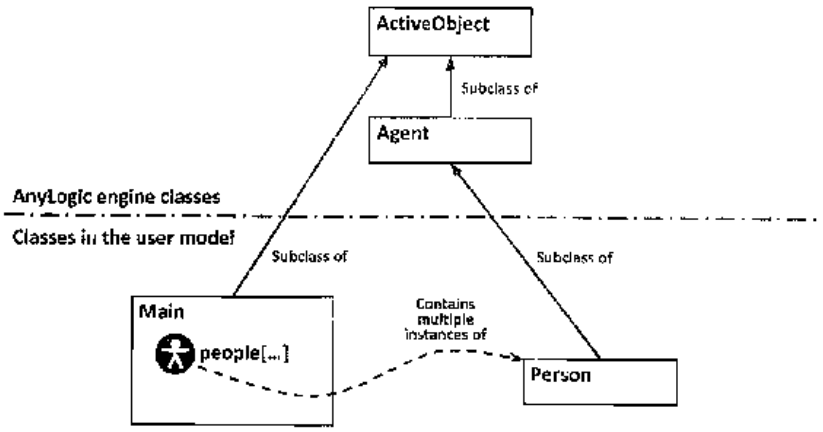


Figure 3.8 UML diagram of a typical agent based model

In AnyLogic you do not need to write code to create subclasses and embed objects into one another; it is done by using drag and drop and wizards. The simplest way of setting up the architecture of an agent based model is to use the **Agent population** wizard.

▶ **To create a population of agents:**

1. Drag the **Agent population** item from the **General** palette to the editor of **Main** (or another container object).
2. In the wizard, specify the name of the agent class (e.g., **Person**), the name of the population (e.g. **people**), the initial number of agents, and, optionally, the space and network settings.

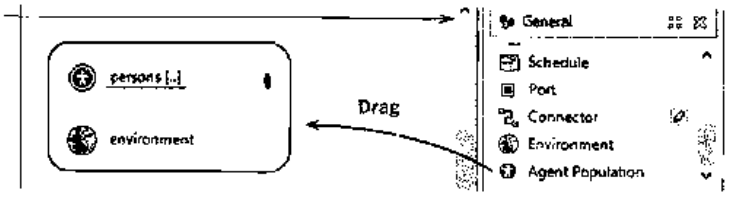


Figure 3.9 The population of agents, the environment, and the animation created by the wizard

There is also a general way of creating a new active object class, making it an agent class, and embedding it into the container class. It includes the following steps.

▶ **To create a new active object class:**

1. Right-click in the **Projects** tree and choose **New | Active object class** from the context menu.
2. Enter the class name (by convention, the first letter of class name should be capitalized) and click **Finish**.
3. The new class appears in the **Projects** tree and its editor opens. Note that the class created this way is not embedded anywhere yet.

▶ **To make an active object class an agent class:**

1. On the **General** page of your active object class properties, check the **Agent** checkbox. This changes the base class from **ActiveObject** to **Agent**. The **Agent** page of the properties becomes available and the icon of the class changes.

▶ **To embed one active object class into another:**

1. Drag the class icon from the Projects tree to the editor of the container object. An embedded object is created inside the container object. So far, there is a single instance.

▶ **To replicate (to create multiple instances of) the embedded object:**

1. Select the embedded object in the editor of the container object and select the Replicated checkbox in its properties. Specify the initial number of objects (by default, the initial number is 0).

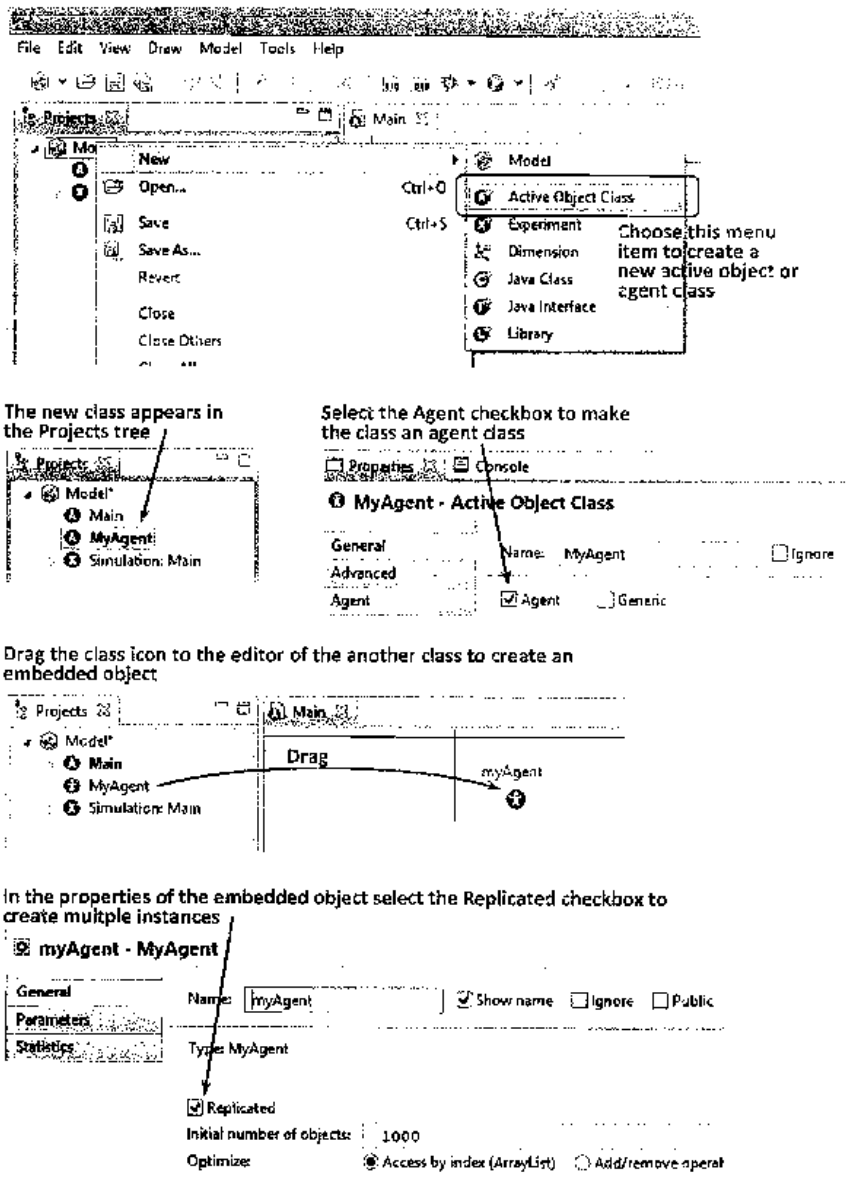


Figure 3.10 The general way of creating a replicated embedded object

3.3. Time in agent based models

When talking about agent based models, we need to distinguish between asynchronous and synchronous time models. *Asynchronous time* means there is no "grid" on the time axis and events may occur at arbitrary moments, exactly when they are to occur. *Synchronous time* assumes things can only happen during discrete time steps (they are "snapped to the time grid"), and nothing happens in between, see Figure 3.11. In synchronous models, on every time step, every agent checks whether it needs to do something.

Do not confuse these time steps with the time steps of the numeric solver that solves continuous-time algebraic and differential equations. The latter are typically smaller and are out of the user control. The user cannot associate an arbitrary action with a numeric step or even know how many times the equations are evaluated.

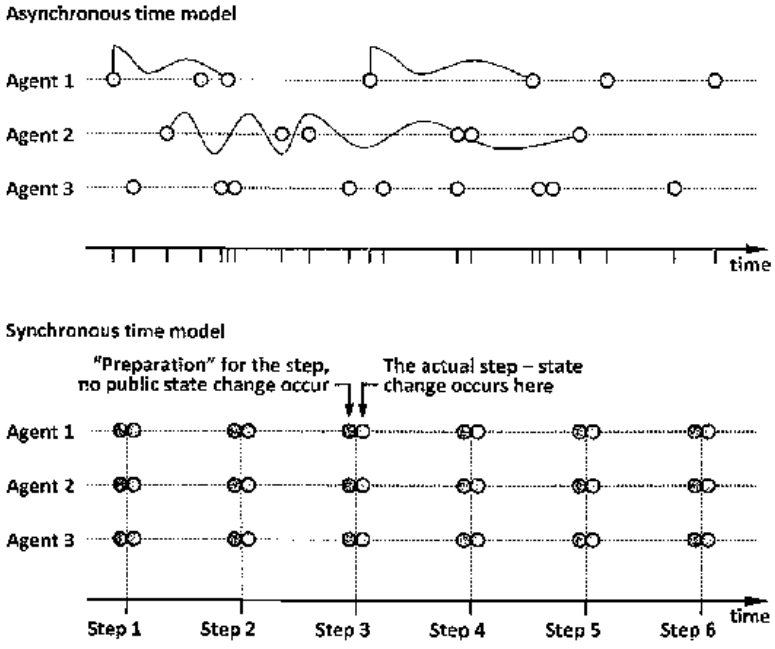


Figure 3.11 Synchronous and asynchronous time in agent based models

Some people think that agent based modeling assumes synchronous time (time steps). This is not true. On the contrary, the most elegant and practically useful

agent based models are either asynchronous or have a mixture of asynchronous time and time steps.

Synchronous time only makes sense in these cases:

- When modeling an artificial world, which is synchronous by definition, such as in "Conway's Game of Life" or in "Schelling segregation" model.
- When the real world system is synchronous. These cases are rare. An example would be a supply chain where inventory decisions are made, for example, on a monthly basis.
- When the agent needs to be updated on what happens around it to recalculate internal variables and check conditions. For example, in (Wallis, Paich & Borshchev, 2004) a person periodically updates his contact rate with other Hispanics, which depends on how many there are in his region. This is, in a way, emulation of continuous time dynamics, typically done at time steps larger than those of numerical solver.

In any case, we recommend using asynchronous time whenever possible, i.e., letting things happen when they really are going to happen. Individual decisions like purchases, job changes, moves, contacts between people, and stochastic events (like breakdowns, recovery, etc.) can just occur at their exact times on the continuous time axis. If there is still a need for time steps, such as in the latter case above, you can mix time steps with continuous time. Example 3.4: "Air defense system" presented later in this chapter is a good example of such a mixture.

Synchronous time inevitably introduces a certain degree of inaccuracy and raises a lot of questions (and thus provides for numerous discussions at "academic conferences" on agent based modeling). Do the results depend on the size of the time step? Do they depend on the order the agents are processed within a time step (deterministic, random)? Can we do a single loop across all agents or do we need to do two loops within a step: the first one to prepare the action and the second one to execute it? In The Game of Life, for example, you cannot do a single loop: all cells must see the previous state of other cells before changing their own state. If a message is sent on a certain time step, on what time step should it be delivered? In addition, synchronous time models are typically computationally less efficient than asynchronous models as every agent is "visited" on every time step, even if it is not doing anything.

Statistics collection in agent based models can also be synchronous and asynchronous. For example, to find out the number of infected people in an epidemic model, you can loop throughout the population and calculate the number at the time you need it, or you can have an always up-to-date counter, which is incremented or decremented by the agents when they change their individual

states. The "synchronous" statistics collection is, however, more standard and common.

3.4. Space in agent based models

Space is widely used in agent based models. Even in the models where location and movement of agents are not important from the model logic viewpoint, space is often used to visualize the agents.

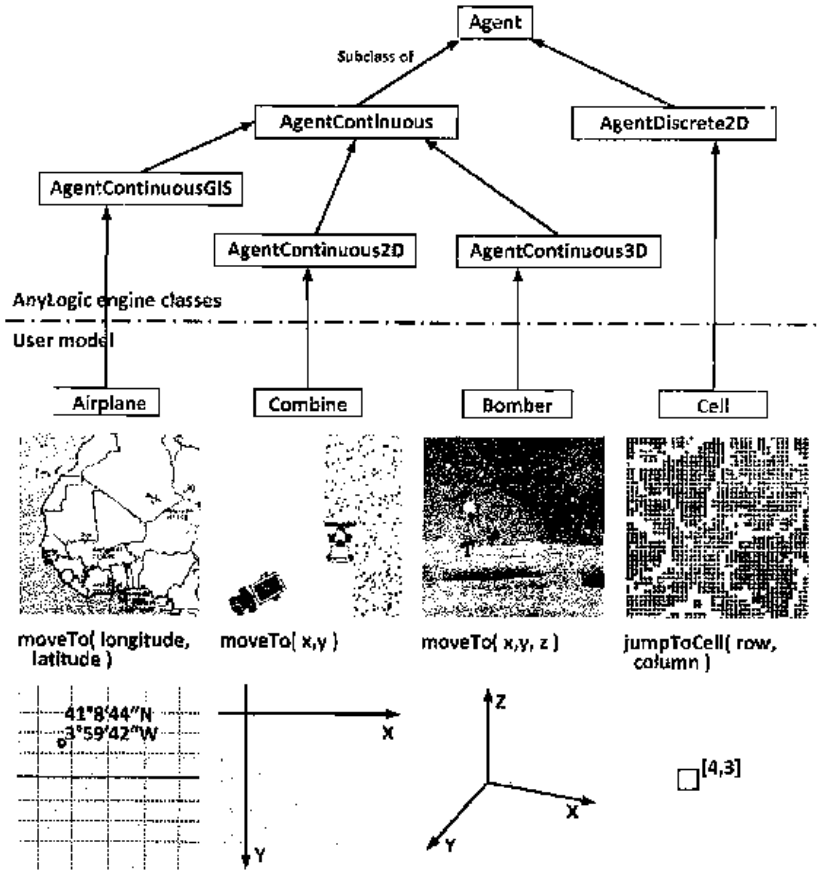


Figure 3.12 Space types in AnyLogic

AnyLogic supports four *types of space*:

- Continuous two-dimensional space
- Continuous three-dimensional space

- Discrete space (grid of cells)
- GIS (Geographic Information System) space

Correspondingly, there are four base classes for agents: **AgentContinuous2D**, **AgentContinuous3D**, **AgentDiscrete2D**, and **AgentContinuousGIS**, see Figure 3.12. Depending on the space type, agents will have different space-related functions and properties, like **moveTo()**, **distanceTo()**, **jumpToCell()**, velocity, column, row, **On arrival** action, etc.

Do not confuse the type of space and the type of animation: 2D continuous or discrete space applies to the model logic only and does not prevent you from creating 3D animation (see Chapter 14) of the model.

The space settings are defined in two places: in the agent itself and in the **Environment** object, see Figure 3.13. The **Environment** object is required for the discrete and GIS space types and is optional for continuous spaces. If the **Environment** object is used, its space type should match the space type of all agents in the environment. If space is irrelevant to the model logic and animation, you can choose arbitrary space settings.

Agents of different types, populations as well as individual agents, can belong to the same environment object and share the same space. Environment can be used to apply layouts to the agents; it also provides some space-related functions, such as **getAgentAtCell()**.

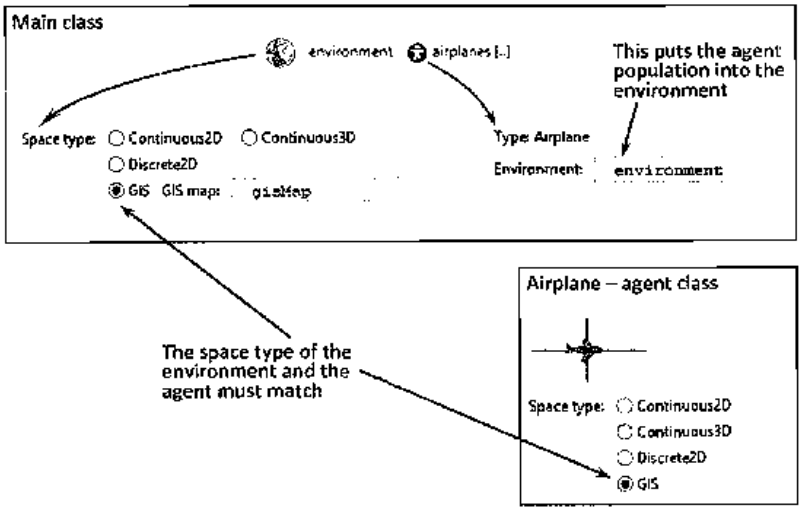


Figure 3.13 Space type settings

3.5. Discrete space

Thinking of agents as cellular automata living in discrete space is another common misconception (the first one is assumption of discrete time). Indeed, there are a lot academic models that use discrete space (for example, The Game of Life ("The Game of Life", n.d.)), Schelling Segregation ("Thomas Schelling ", n.d.)), Heat Bugs, etc.), but in most industrial-level models, space-aware agents live and move in continuous 2D or 3D space.

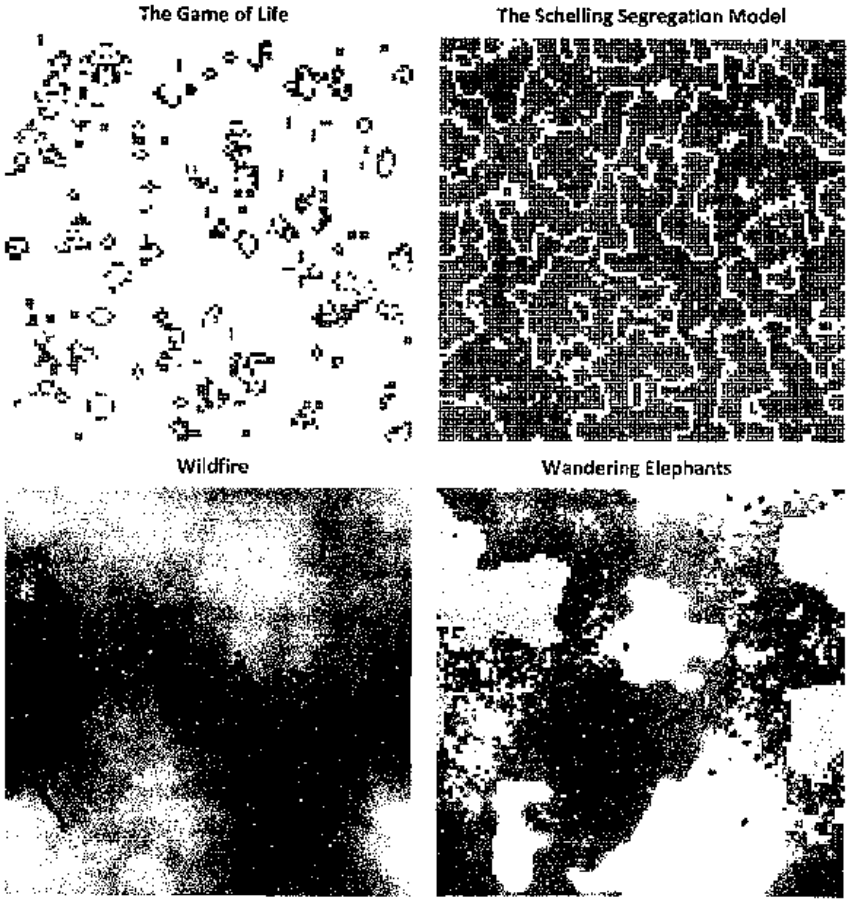


Figure 3.14 Examples of discrete spaces

With that said, we will, nevertheless, spend some time on discrete space because there are cases when grid cells are natural and useful. For example, they can be

used to model dynamics of vegetation, water resources, wildfire, or population density in a geographical space. In such models, land is partitioned in square cells with each cell being an agent with its own variables and behavior. Cells may interact with each other and with other types of agents, for example, with agents freely moving in 2D space that overlaps the discrete space, see Example 3.3: "Wildfire" described later in this section.

Discrete space is a rectangular grid of cells, see Figure 3.14. In AnyLogic, it is created under the control of the **Environment** object.

▶ **To create an agent population living in discrete space:**

1. Drag the **Agent population** object from the **General** palette to the graphical editor.
2. In the **Configure new environment** page of the wizard choose **Space type: Discrete 2D** and set up the space size.

The options are shown in Figure 3.15. You can choose the visual size of the entire space and the number of rows and columns. A cell is, in general, a rectangle whose width equals the space width divided by the number of columns, and height is the space height divided by the number of rows. AnyLogic does not draw any grid lines or rectangles for cells so it is up to you to decorate the space.

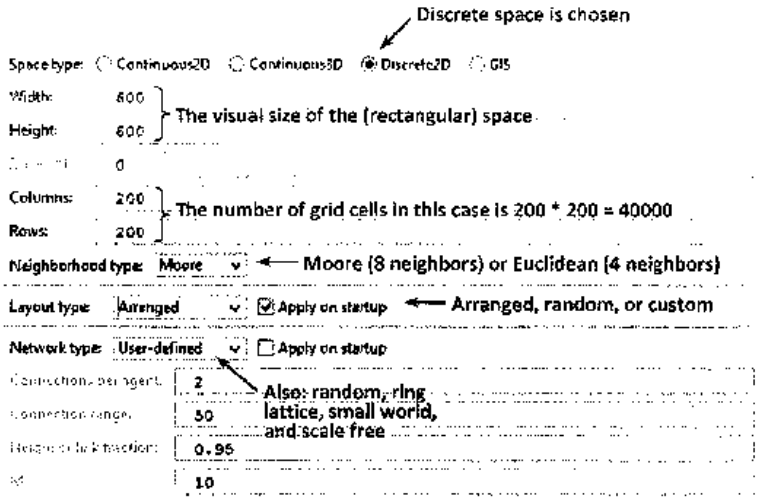


Figure 3.15 Discrete space options (Advanced page of the Environment properties)

The position of the discrete space rectangle on the canvas of the container object, namely its top left corner, is defined by the position of the presentation of the discrete space agent, see Figure 3.16. Therefore, if there are multiple populations of

agents in a single discrete space environment, their presentation shapes should be located at the same point.

A grid cell may be either empty or occupied by, at most, one agent. In many models there is an agent in every cell. In other words, *a cell is itself an agent*. In such models, agents typically do not move. In other models (e.g. in Example 3.1: "Schelling segregation"), there are fewer agents than cells and agents can move from one cell to another. In the latter case you can apply different layout types to the agents, such as random or arranged.

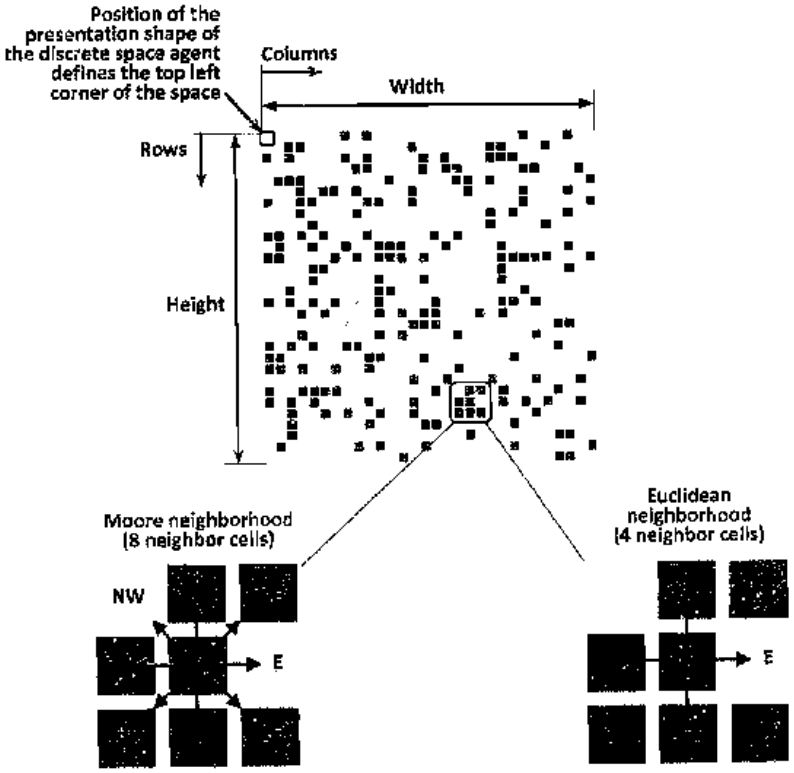


Figure 3.16 Dimensions and neighborhood types of the discrete space

In discrete space there is a notion of *neighborhood*. There are two types of neighborhoods: Moore and Euclidean (also known as Von Neumann). In the *Moore neighborhood* a cell has eight neighbors, and the *Euclidean neighborhood* a cell has four neighbors, see Figure 3.16. The neighborhood type will affect the result of the `getNeighbors()` function. The constants `NORTH`, `NORTHEAST`, `EAST`, etc. defined in the `Agent` class are used to address directions in the discrete space.

Although networks are not frequently used in conjunction with discrete space, a choice of social networks is available.

Example 3.1: Schelling segregation

In the 1970s, Thomas Schelling, a Nobel prize winning economist, showed in his articles dealing with racial dynamics "that a preference that one's neighbors be of the same color, or even a preference for a mixture "up to some limit", could lead to total segregation, thus arguing that motives, malicious or not, were indistinguishable as to explaining the phenomenon of complete local separation of distinct groups. He used coins on graph paper to demonstrate his theory by placing pennies and nickels in different patterns on the "board" and then moving them one by one if they were in an "unhappy" situation." ("Thomas Schelling ", n.d.)).

We will implement Schelling's model as a discrete space/discrete time agent based model. The space represents a city, and each cell represents a house. Agents are people and are two colors: yellow and red. Initially, people are randomly distributed across the city. There are fewer people than houses, so there is always the possibility that a person could move. The agent behavior in this model is very simple:

- If the percentage of people of the same color among one's neighbors is lower than a certain threshold value, the person feels unhappy and moves to a randomly chosen empty house; otherwise the person is happy and does nothing.

The threshold value (the preference for the same color) will be a parameter of the model. Discrete time is originally assumed in this model: the evaluation of happiness and moves are performed on discrete time steps. It is, however, possible to implement an asynchronous version of the model.

▶ **Create the population of agents and discrete space:**

1. Create a new model and drag the **Agent population** object from the **General** palette to the editor of **Main**.
2. In the first page of the wizard, set the **Population name** to **people**, the **Initial population size** to **8000**, and choose **rectangle** as **Animation**.
3. In the next page of the wizard, choose the **Discrete 2D** space type and click **Finish**.
4. Open the editor of **Person**, and add a new parameter. Set the parameter name to **color** and choose **Color** as the type of the parameter.
5. Type the following expression in the default value field of the parameter: **randomTrue(0.5) ? red : yellow**. This will ensure there is approximately the same number of red and yellow people in the model.

6. Select the rectangle shape at the coordinate origin of **Person** and type color in the **Fill color** field of its **Dynamic** property page.
7. Run the model. See the red and yellow people randomly distributed across the discrete 100 by 100 space, see Figure 3.17, on the left.

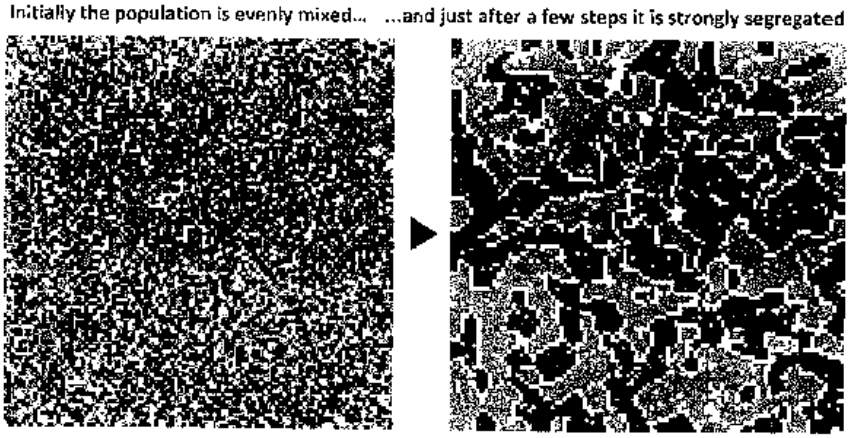


Figure 3.17 The Schelling Segregation Model. Preference is set to 60%

▶ **Enable the discrete time in the model and implement agent behavior:**

8. In **Main**, create a new parameter **Preference** of type **double** and default value 0.6. This will be the minimum threshold value for the percentage of same color neighbors.
9. Open the editor of **Main**, select the **environment** object, and select **Enable steps** in its properties. This enables time steps and now we can type behavior rules in the **On before step** and **On step** action fields of the agents.
10. Return to the editor of **Person** and add a Boolean variable **happy** with initial values **true**.
11. Open the **Agent** page of **Person** properties and type the following code in the **On before step** field:

```
Agent[] neighbors = getNeighbors(); //default neighborhood model is Moore
if( neighbors == null ) {
  happy = true; //no neighbors are good neighbors
} else {
  //count same color neighbors
  int nsamecolor = 0;
  for( Agent a : neighbors )
    if( ((Person)a).color.equals( color ) ) //need to cast generis Agent to Person
      nsamecolor++;
  //compare with minimum number
  happy = nsamecolor >= neighbors.length * get_Main().Preference;
```

```
}
```

12. And in the field **On step** type:

```
if( !happy )
  jumpToRandomEmptyCell();
```

13. Run the model. Watch how the initially mixed population becomes almost fully segregated in just a few steps; see Figure 3.17, on the right.

Even a slight preference to have at least 60% of neighbors of the same color leads to strong segregation. You can link a slider control to the **Preference** parameter and experiment with different values. Another implicit parameter of the model is the occupancy of the space. In our case it is 80% because we have 8,000 people and 10,000 cells (houses). Occupancy affects the ability of the system to reach equilibrium at high values of preference.

Some comments on the model implementation. Notice that in this model each step has two phases: preparation and action (this scheme is shown in Figure 3.11). An agent evaluates its neighborhood in the preparation phase (**On before step**) and moves in the action phase (**On step**). The agent's internal variable **happy** is only needed to keep the neighborhood evaluation results between the two phases.

The two-phase steps are necessary in this case because we must ensure people do not move before everyone has evaluated their current neighborhood. If agents were doing both things at once, some people would be evaluating their current neighborhood and some would be evaluating "intermediate" neighborhoods in the middle of moving, which does not make any sense.

? Convert this model into an asynchronous model. Instead of global steps you can use a cyclic event with random recurrence time inside each agent. In the asynchronous model you will not need to separate neighborhood evaluation and moving. Compare the simulation results.

Example 3.2: Conway's Game of Life

And of course we will show how to program, in AnyLogic, The Game of Life – the famous invention of John Conway that "opened up a whole new field of mathematical research, the field of cellular automata" ("Conway's Game of Life", (n.d.)). The universe of The Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states: dead or alive. Every cell interacts with its eight neighbors (Moore neighborhood model is assumed). At each step:

- Any live cell with fewer than two live neighbors dies, as if caused by under-population.

- Any live cell with two or three live neighbors lives on to the next generation.
- Any live cell with more than three live neighbors dies, as if by overcrowding.
- Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The initial pattern constitutes the seed of the system, and the rules are applied repeatedly to create further generations " ("Conway's Game of Life", (n.d.)).

The following is interesting in the context of the agent based modeling technique. One may try to model a dead cell as a cell with no agent in it, and a live cell as a cell with an agent. This however will lead to a very awkward algorithm of reproduction. A more elegant implementation is having, in every cell, an agent with two possible states: dead or alive.

▶ **Create discrete space and cells:**

1. Create a new model and drag the **Agent population** object from the **General** palette to the editor of **Main**.
2. In the first page of the wizard, set the **Agent class name to Cell, Population name** to **cells**, the **Initial population size to 10000**, and choose **rectangle** as **Animation.**
3. In the next page of the wizard, choose the **Discrete 2D** space type and click **Finish.**
4. Open the editor of **Cell** and add a Boolean variable **alive** with initial value **randomTrue(0.2)**. 20% of cells will be randomly chosen to be initially alive.
5. Select the rectangle (the cell animation shape created by the wizard) and type this code in the **Fill color** field of its **Dynamic** property page: **alive ? mediumBlue : lavender.**
6. Run the model. See the initial random pattern.

▶ **Program the cell behavior:**

7. Open the editor of **Main**, select the **environment** object, and select **Enable steps** in its properties. This enables time steps and now we can type behavior rules in the **On before step** and **On step** action fields of the agents.
8. Return to the editor of **Cell** and add an integer variable **naliveneighbors.** This variable will keep a count of live neighbor cells between the evaluation and action phases of the step, similar to the **happy** variable in Example 3.1: "Schelling segregation".
9. Open the **Agent** page of **Cell** properties and type the following code in the **On before step** field:

```
naliveneighbors = 0; //reset counter
for( Agent a : getNeighbors() ) //count all alive neighbors
  if( ((Cell)a).alive ) //need to cast generic Agent to Cell
    naliveneighbors++;
```

10. And in the field **On step** type:

```
if( alive && naliveneighbors < 2 )
  alive = false; //die because of loneliness
else if( !alive && naliveneighbors == 3 )
  alive = true; //new cell is born
else if( alive && naliveneighbors > 3 )
  alive = false; //die because of overcrowding
```

11. Run the model and watch the evolution of this exciting artificial world.

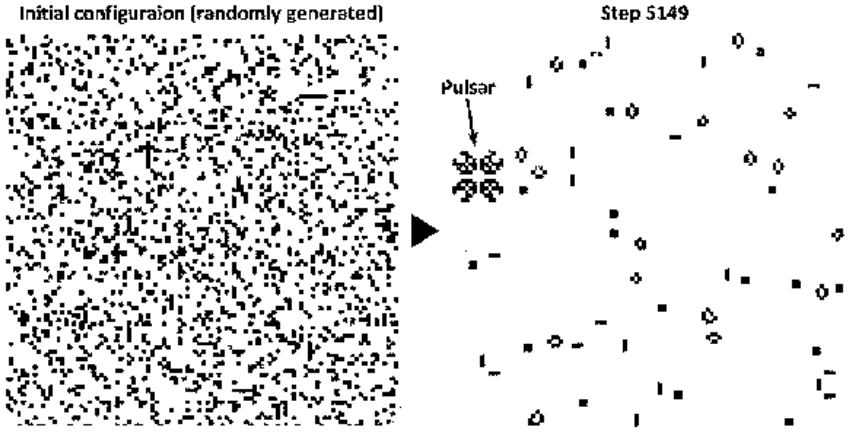


Figure 3.18 The Game of Life

Although The Game of Life is a "zero-player game", i.e., its evolution is entirely determined by its initial state, we can bring in some instructiveness. For example, we can program the cell to toggle its state on mouse click. Then we will be able to initiate additional waves of evolution in static or oscillating configurations.

▶ **Add cell's reaction on mouse click:**

12. In the editor of **Cell**, select the rectangle animation shape and type this code in the **On click** field of its **Dynamic** property page: **alive = ! alive;**
13. Play with the model.

Example 3.3: Wildfire

We will build a model of a wildfire. In this example we will show how the two types of space, continuous and discrete, can be linked.

We will model vegetation as grid cells – agents in discrete space. The burning time of a cell is proportional to the amount of "fuel" in the cell, which will be randomly generated at the model startup. While burning, the cell may cause ignition in the

adjacent cells. The ignition may also be caused by a bomb dropped by an aircraft – an agent moving in continuous 2D space that overlaps the discrete space.

As an additional exercise, we can introduce the wind into the model and make the probability of ignition depend on the wind direction.

▶ **Create the grid cells:**

1. Create a new model and drag the Agent population object to the editor of Main. On the first page of the wizard type:
   Agent class name: GridCell
   Population name: gridcells
   Initial population size: 40000
   Animation: Rectangle
   Press **Next**.
2. On the second page of the wizard, choose the **Discrete 2D Space type** for the environment and make the following settings:
   Width: 600
   Height: 600
   Columns: 200
   Rows: 200
   Initial location: Arranged
   Press **Finish**. The wizard creates a new environment object, a population of grid cells, and places the animation of a cell (a small blue rectangle) nearby.
3. Run the model. You should see a square space filled with 40,000 cells. You should also see that the model consumes 60-70% of the default available memory (64K). This is because of the number of agents.
4. Open the editor of **GridCell**. Select the blue rectangle and change both its width and height to 3 pixels (these settings are on the **Advanced** property page). Now the cell animations will not overlap.
5. Open the editor of the **Simulation** experiment and extend the frame (the default model window border) to 800 by 650 pixels. The entire area now will be visible.
6. Open the **Advanced** page of the **Simulation** experiment and set **Maximum available memory** to 512M. This will allow us to further add internal memory to our agents.

The next step is to create the initial distribution of fuel in the cells. We could use real geographical data, but in this simple model we will generate a random landscape with forests and deserts.

▶ **Model the initial vegetation distribution:**

7. Open the editor of GridCell and add a parameter Fuel of type double. This will be the amount of fuel in the cell.
8. Go to the editor of Main and add a function makeUpInitialFuel(). Type this code in the function body:

```
int N = environment.getColumns();
//generate forest "seeds"
for( int n = 0; n < 200; n++ ) { //200 seeds
  int x = uniform_discr( 0, N - 1 );
  int y = uniform_discr( 0, N - 1 );
  for( int k = 0; k < 1000; k++ ) {
    int i = ( x + (int)triangular(-50,0,50) + N ) % N; //%N to stay witnin the space
    int j = ( y + (int)triangular(-50,0,50) + N ) % N;
    ((GridCell)( environment.getAgentAtCell( i, j ) )).Fuel += 0.25;
  }
}
//smooth
for( int n = 0; n < 10; n++ ) {
  for( GridCell gc : gridcells ) {
    double sum = gc.Fuel;
    for( AgentDiscrete2D ad : gc.getNeighbors() )
      sum += ((GridCell)ad).Fuel;  //getNeighbors() returns generic agent class
    gc.Fuel = sum / (gc.getNeighbors().length + 1);
  }
}
//find min/max fuel values
double min = +infinity;
double max = -infinity;
for( GridCell gc : gridcells ) {
  double f = gc.Fuel;
  if( f > max ) max = f;
  if( f < min ) min = f;
}
//normalize into [-0.2,1.3]
for( GridCell gc : gridcells ) {
  gc.Fuel -= min; //to [0..max-min]
  gc.Fuel *= 1.5 / ( max - min ); //to [0..1.5]
  gc.Fuel -= 0.2; //to [-0.2..1.3]
}
//update cells
for( GridCell gc : gridcells )
 gc.rectangle.setFillColor( lerpColor( gc.Fuel, paleGoldenRod, new Color(65, 100, 0) ) );
```

9. In the Startup code of Main type: makeUpInitialFuel(); This will call the function at the model startup to make up the initial landscape.
10. Run the model. The area covered with the cells should look like the one shown in Figure 3.19.

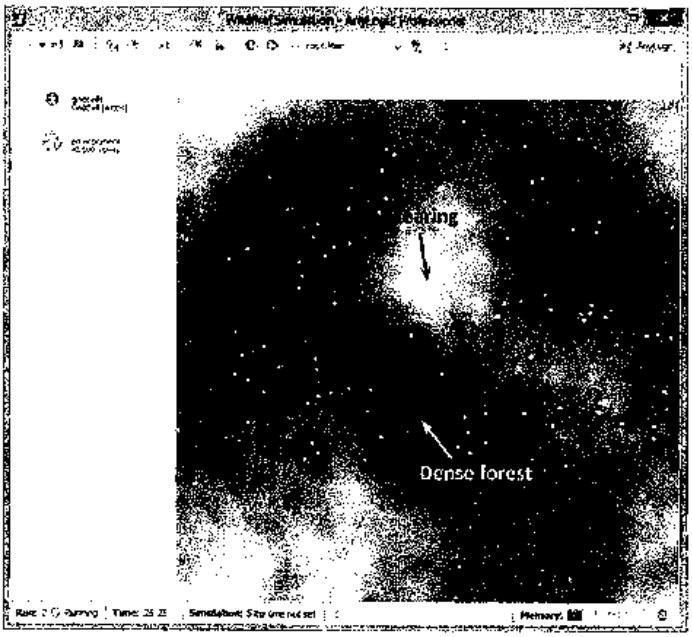


Figure 3.19 A randomly generated landscape

The landscape generation algorithm we used is just one of many possibilities. It starts with several "forest seeds" randomly distributed all over the area. Then it creates forests around those seeds by increasing the amount of fuel (the closer to the seed, the larger the increase). We then normalize the values of the cell fuel to the interval [-0.2,1.3] and modify the initial color of the cell animation to reflect the value of the **Fuel**.

Now we will program the cell behavior. In the first version of our model the fire diffusion will not depend on the wind.

▶ **Program the cell behavior (the fire diffusion):**

11. In the editor of **GridCell** create the statechart, as shown in Figure 3.20.
12. Select the cell animation (the rectangle) and type this code in the **On click** field of its **Dynamic** property page: **statechart.receiveMessage("Ignition");** This will allow us to manually initiate the wildfire.
13. Run the model and try clicking in the areas of various fuel densities. See how the wildfire spreads.

The behavior of a grid cell needs detailed explanation. The cell can be in one of three states: **Normal**, **Burning**, and **BurnedOut**. The transition from **Normal** to **Burning** is triggered by the message "Ignition", which may come from an adjacent cell, or from a mouse click (later on, when we add an aircraft, it will be coming from the aircraft as well). The burning time of a cell is set to the value of the **Fuel** parameter (see the transition **OutOfFuel**): the more wood that is in the cell, the longer the cell will burn.

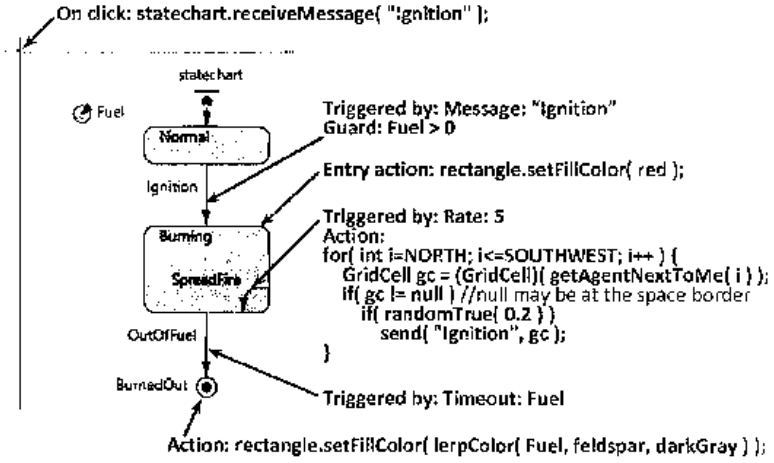


Figure 3.20 A vegetation grid cell behavior and a wildfire caused by a mouse click

The most interesting part of the statechart, however, is the transition **SpreadFire**. First of all, this transition is internal (see Section 7.3) to the state **Burning**, which means its execution will not reset the timeout of the transition **OutOfFuel**. **SpreadFire** will be executed at the rate of 5, i.e., on average 5 times per time unit. Therefore, the longer the cell is burning, the more occurrences of **SpreadFire** there will be. Upon each occurrence, the cell iterates across all neighbor cells. (In the **for** loop we use the knowledge about the values of the direction constants: **NORTH** is 0, **SOUTH** is 1, ..., **SOUTHWEST** is 7. This knowledge can be obtained from AnyLogic API reference.) The message "Ignition" is sent to each of the neighbor cells with 20% probability.

Try to increase the real time scale and see how fast the model is. This model is computationally very efficient because it is *completely asynchronous*; there are *no time steps*. There are quite a few cells (40,000), but inactive cells do not consume any CPU time so there is no global loop across all cells.

Our next step is to model the aircraft that drops bombs as it flies over the area. The aircraft will be an agent living in continuous 2D space that overlaps with the discrete space. This time we will not be using the Agent population wizard because there will be only one aircraft. It will live in 2D space and will not need an environment object.

▶ **Create the Aircraft agent class:**

14. Right-click the model (top-level) item in the **Projects** tree and choose **New | Active object class** from the context menu. Name the class **Aircraft** and press **Finish**.
15. Select the **Agent** checkbox in the **Aircraft** class properties.
16. Drag the **Plane** picture from the **Pictures** palette to (0,0) of the **Aircraft** editor. Rotate the picture so that the aircraft looks towards the right (this side of the agent moves forward), and reduce the size by 50%.
17. Open the **Agent** page of the **Aircraft** properties and make sure the space type is Continuous 2D. On the same page set the velocity of the aircraft to 30.
18. Return to the editor of **Main** and drag the **Aircraft** class icon from the **Projects** tree there. The animation of the airplane appears at (0,0) of **Main**. Move the aircraft animation to the same point where the cell animation is located. This will synchronize the animation locations.

▶ **Let the aircraft fly over the area:**

19. Type this code in the Startup code field of the aircraft:
```
setXY( 0, uniform( 400, 600 ) ); //set the initial location
moveTo( 600, uniform( 0, 200 ) ); //start moving towards the right
```
The initial position is chosen randomly at the west edge of the area, closer to the south, and the final position is on the east side, closer to the north.
20. Run the model and see the aircraft flying.

Now we will model dropping the bombs. This is exactly where the two types of space will be linked in our model.

Figure 3.21 Wildfire caused by bombing

▶ **Model bombing:**

21. Return to the editor of the **Aircraft** class and add an **Event** object. Name the event **bomb**. Set the event **Mode** to **cyclic** and set **Recurrence time** to 5.
22. Type this code in the **Action** field of the event:

```
EnvironmentDiscrete2D env = get_Main().environment;
double cellw = env.getWidth() / env.getColumns(); //cell width in pixels
double cellh = env.getHeight() / env.getRows(); //cell height in pixels
//transalate (X,Y) into (row,column)
int c = (int)( getX() / cellw );
int r = (int)( getY() / cellh );
//send the message to the cell at that position
env.getAgentAtCell( r, c ).receive( "Ignition" );
```

23. Open the **Agent** page of the **Aircraft** properties and type this code in the **On arrival** field: **bomb.reset();** This will turn off the cyclic **bomb** event when the aircraft has reached the edge of the area.
24. Run the model.

While the aircraft is flying, it drops a bomb every 5 time units. The bomb ignites the cell right below the aircraft. The translation of the continuous coordinates (X,Y) into the discrete coordinates (row, column) is done using the grid cell width and height.

? Modify the model to take wind into consideration. Make the probability of passing the fire from cell to cell depend on the wind direction, as shown in Figure 3.22. Assume the wind direction can take eight values {NORTH, NORTHEAST, ..} and the strength of the wind is always the same.

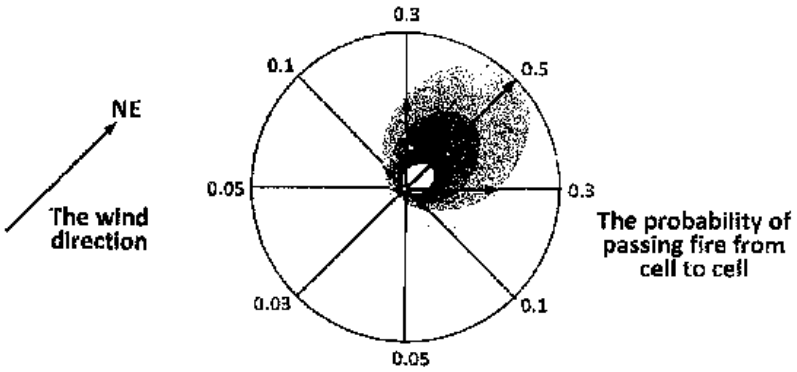


Figure 3.22 The probability of passing the fire depending on the wind direction

Tip: Program the function that converts the direction constant to angle, and then another one that compares two angles and returns the probability. Use the probability in the action of the SpreadFire transition instead of 0.2.

Discrete space API

The base class for agents that live in discrete space is **AgentDiscrete2D**, which is a subclass of **Agent**, which, in turn, is a subclass of **ActiveObject**. In addition to the functionality common to all kinds of agents, **AgentDiscrete2D** offers the following API:

- **int getC()** – returns the column of the agent's cell.
- **int getR()** – returns the row of the agent's cell.
- **jumpToCell(int r, int c)** – moves the agent into a cell with the given row and column.
- **setCell(int r, int c)** – puts the agent into a given cell (initialization use only).
- **boolean jumpToRandomEmptyCell()**– finds a random empty cell and places the agent there.
- **moveToNextCell(int dir)** – moves the agent to an adjacent cell in a given direction.
- **swapWithAgent(AgentDiscrete2D anotherAgent)** – swaps the cell location of this agent with another agent. Swap is the only way for an agent to move in a fully occupied space.

- **swapWithCell(int r, int c)** – swaps this agent with an agent at the cell with the given row and column.
- swapWithNextCell(int dir) – swaps the agent with an agent at the adjacent cell in a given direction.
- int[] findRandomEmptyCell() –tries to find a randomly located empty cell and returns its row and column in the array with two elements.
- Agent **getAgentAtCell(int r, int c)** – returns the agent located in the cell with a given row and column, or null.
- Agent **getAgentNextToMe(int dir)** – returns the agent next to this agent in a given direction, if any.
- **AgentDiscrete2D[] getNeighbors()** – returns the array of neighbor agents, subject to the current neighborhood type (Euclidean {N, S, E, W}, Moore - also {..,NW, NW, SE, SW}). The elements in the returned array are of the base class **AgentDiscrete2D**, and not of the same class as this agent because, in general, there can be more than one type of agent in the same space.

In addition, the environment object supporting 2D continuous space (base class **EnvironmentDiscrete2D**) offers these functions:

- int[] findRandomEmptyCell() – tries to find a randomly located empty cell and return its row and column in the array with two elements.
- **AgentDiscrete2D getAgentAtCell(int r, int c)** – returns the agent located in the cell with a given row and column, or null.
- **int getColumns()** – Returns the number of columns in the space.
- **int getRows()** – Returns the number of rows in the space.

3.6. Continuous 2D and 3D space

Continuous space (two- or three-dimensional) in AnyLogic is infinite space with real number coordinates (Java type for coordinate is **double**). Continuous 2D space is assumed by default when a new agent class is created. If you want to apply standard layouts or networks to the agents, you should use the **Environment** object. Otherwise, the **Environment** object can be omitted in the model.

▶ **To create an agent population living in continuous 2D or 3D space:**

1. Drag the **Agent population** object from the **General** palette to the graphical editor.
2. In the **Configure new environment** page of the wizard choose **Space type**: **Continuous 2D** or **Continuous 3D** and choose the width and height, and the initial layout.

The options of the **Environment** object related to the continuous space are shown in Figure 3.23. The **Width** and **Height** are *not the bounds of the entire space*; the space is always infinite and unbounded. These are the bounds of the layout, and they apply each time the layout is applied (in most cases, the layout is applied once, when the model initializes). The agents can freely cross these bounds when they move.

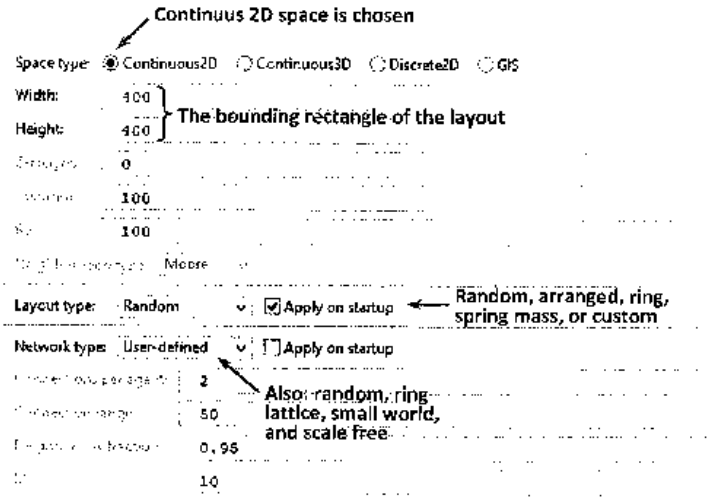


Figure 3.23 Continuous 2D space options (Advanced page of the Environment properties)

When you create an agent population, or when you drop an individual agent on the canvas of the container object, its animation is also placed there. The design-time position of the animation defines the point where the agent with coordinates (0,0) will be displayed at runtime (see Figure 3.24), but *does not affect the actual coordinates of the agent.*

If there is more than one agent population or individual agent and, correspondingly, more than one embedded agent animation, it makes sense to place all the animations into the same point to make sure that at runtime the positions of the agent animations are consistent with their coordinates.

To see the agent animation in the 3D scene, make sure its **Show in 3D scene** checkbox is selected.

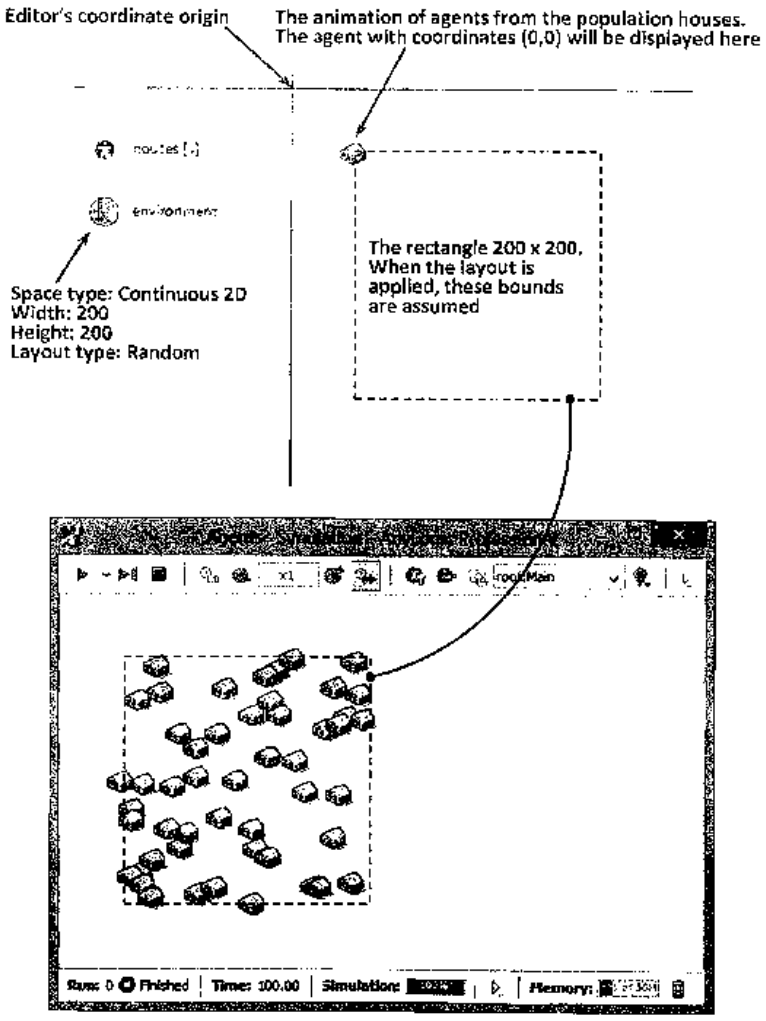

Editor's coordinate origin
The animation of agents from the population houses. The agent with coordinates (0,0) will be displayed here
The rectangle 200 x 200. When the layout is applied, these bounds are assumed
Space type: Continuous 2D
Width: 200
Height: 200
Layout type: Random

Figure 3.24 The meaning of the width, height, and the layout properties

Movement in continuous space

The agent movement is always *piecewise linear*: the agent moves along straight line segments at constant velocity. To initiate a straight-line movement from the current position to the point with coordinates (Xb,Yb), you should call the function `moveTo( Xb, Yb )`, in 3D space this is `moveTo( Xb, Yb, Zb )`. Optionally, you can suggest a polyline-based trajectory for the agent movement by providing the polyline as the last

parameter of the moveTo() function, e.g. moveTo(x, y, z, polyline), see Figure 3.25. The agent will first move from its current location to the closest point on the polyline along the shortest straight line, then move along the polyline to the point on the polyline closest to the destination, and then straight to the destination point. You can model movement along curved trajectories by approximating them by polylines.

The agent stops when it reaches the destination point, or when stop() or jumpTo() is called. You can also call moveTo() another time while the agent is moving, and it will then change its direction.

The velocity of the agent is set at the bottom of its Agent property page and can be changed at runtime by calling the function setVelocity(). If this function is called when the agent is moving, it continues to move with the new velocity. To model acceleration/deceleration you can break down the movement into small segments and increase or decrease the agent velocity at the beginning of a segment.

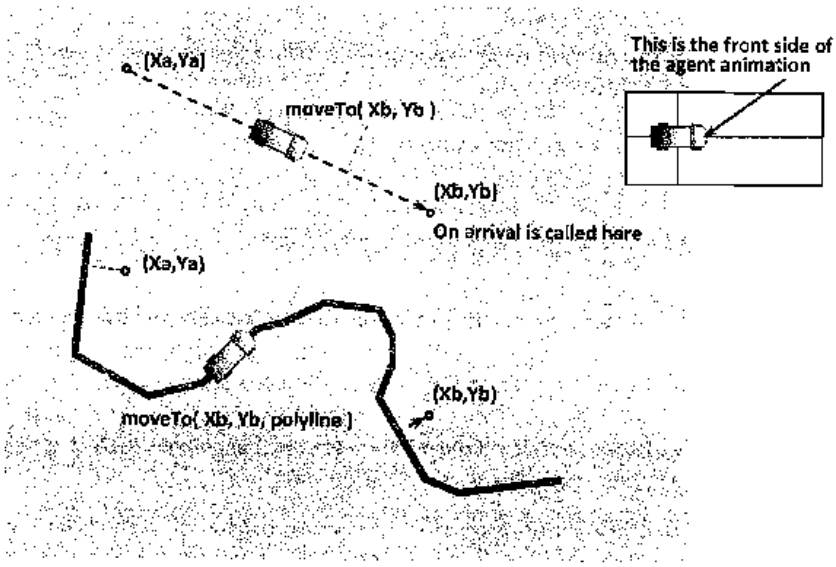


Figure 3.25 Movement in continuous 2D space

When an agent moves, the right-hand side of its animation is the front side. To change the front side, you need to rotate the animation.

When the agent successfully reaches the destination point, its On arrival code (specified in the Agent property page) is executed, and active statecharts transitions waiting on arrival are triggered, if there are any. The call of moveTo() followed by an arrival-triggered transition is a frequently used pattern.

To calculate the distance to another agent or to a point you can use the functions distanceTo(x, y), distanceTo(x, y, z), and distanceTo(agent).

To detect the moment when a geometric condition over the moving agents becomes true (for example, when two agents get close to each other), you can use two approaches:

- Analytical. You can obtain the time analytically using your high school knowledge of physics and geometry, and then schedule the required action at the exact moment of time with the help of event or transition. The solution will be 100% accurate and very efficient computationally.
- Test the condition periodically (polling). You can introduce time steps ("ticks") and test the condition on each tick. Such a model will consume a lot more CPU power, and the detection will occur with an error (the larger the time step, the larger the error), or can be even missed. However, this approach is simpler and also more general.

The example Air defense system presented below demonstrates most of continuous space modeling techniques.

Example 3.4: Air defense system

We will build a simple model of a radar-based air defense system. All types of agents in this model (bomber aircrafts, radars, missiles, bombs, and buildings) live and interact in continuous 3D space. We will use various kinds of movement and space sensing techniques.

Bombers are sent to destroy ground assets (buildings) compactly located in a certain area. One aircraft is launched per building. Aircrafts carry bombs. To complete its mission successfully, an aircraft needs to drop a bomb within 500 meters ground distance from the target building while flying, at most, at 2 km altitude. Having completed the mission, the bomber returns to the base using a higher altitude route. The bomber speed is 600 km/h.

The buildings are protected by the air defense system, which consists of two radars equipped with guided surface-to-air missiles. A radar can simultaneously guide up to two missiles. A missile is launched once the bomber enters the radar's coverage area, which is a hemisphere of 6.5 kilometer radius around the radar. The missile speed is 900 km/h. The missile explodes once it gets as close as 300 meters to the aircraft. If the missile exits the radar coverage area before it hits the aircraft, it destroys itself.

▶ **Create the scene:**

1. Create a new model. Drag a **Rectangle** shape into the editor of **Main**, place its top left corner at (0,0), and set its size to 800 x 600. We will assume 10 pixels per kilometer scale.
2. Set the **Line color** of the rectangle to **No color** and use **Earth texture** for the **Fill color**.
3. Check the **Show in 3D scene** checkbox on the **General** page of the rectangle properties. On the **Advanced** property page set the rectangle's Z to -1 and Z-height to 1.
4. Finally, on the **General** page check the **Lock** checkbox. This prevents the rectangle from being selected and we will be able to freely draw on top of it.
5. Open the **3D** palette and drag the **3D window** to the editor of **Main**. Place the window below the ground rectangle, for example, at (0,850) and extend it to the size of 800 by 550 pixels.
6. From the **Presentation** palette drag the **View area** and place it at (0,800). Set its **Name** to **view3D** and **Title** to **3D**.
7. Create another view area at (0,0) with the name **view2D** and title **2D**.
8. Run the model. Switch between the 2D and 3D views.

▶ **Create the buildings:**

9. Draw a closed polyline on top of the ground rectangle, as shown in Figure 3.26. Name it **protectedArea**. Our buildings will be located within its bounds.
10. Drag the **Agent population** object from the **General** palette to **Main** (it makes sense to place it to the left of the ground rectangle so it does not interfere with the animation).
11. On the first page of the wizard, type:
    Agent class name: Building
    Population name: buildings
    Initial population size: 10
    Press **Next**.
12. On the second page of the wizard, choose the **Continuous3D Space type** for the environment. Press **Finish**. The wizard creates a new environment object, a population of buildings, and places the animation of the building nearby, see Figure 3.26.
13. Drag the animation of asset to (0,0) to synchronize the coordinate systems of the agents (buildings) and the **Main** object.
14. Run the model. See 10 animation shapes of the buildings (so far a person shape is used, but we will change it later) randomly distributed in a cubicle 400 by 400 by 400 pixel space. (This was the default space size setting in

the environment object. We do not need to change it as we will customize the asset locations.)

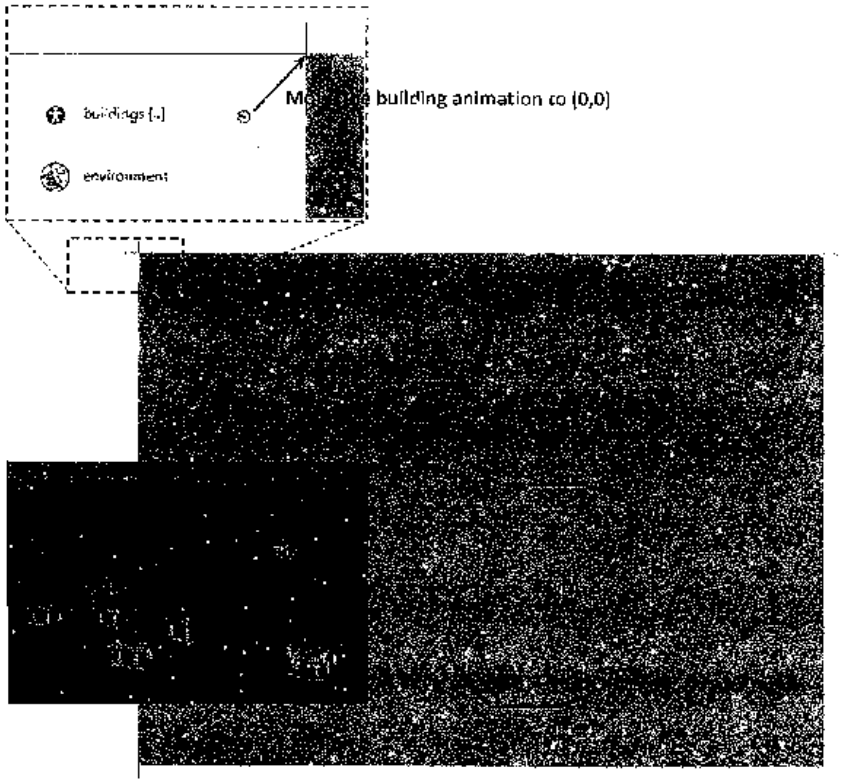


Figure 3.26 The scene of the Air Defense System model

In the next phase, we will change the animation of building to 3D **House** object and place assets on the ground within the **protectedArea**.

▶ **Change the animation of building and set the buildings locations:**

15. Open the editor of **Building**, delete the animation created by the wizard, and drag the **House** object from the **3D Objects** palette to (0,0). Set the **Scale** of the object to 25%.
16. In the **General** page of the **Building** properties write the following **Startup code**:
    ```
    Point pt = get_Main().protectedArea.randomPointInside();
    setXYZ( pt.x, pt.y, 0 );
    ```
    This will place the house on the ground and within the polyline bounds.

17. Run the model and see where the buildings are located. Note that the polyline **protectedArea** is visible in the 2D view, but not in the 3D view. This is because we have not selected its **Show in 3D scene** property.

Now we will create the bomber aircrafts and let them fly to the buildings and return to the base.

▶ **Create the Bomber agent class:**

18. Return to the editor of **Main** and create another **Agent population** (place it nearby the environment object to the left of animation). In the wizard, make these settings:
    Agent class name: Bomber
    Population name: bombers
    Initial population size: 0
    Environment: Use existing (environment)
    Press **Finish**.
19. In the editor of **Main**, select the default animation of the bomber (it should be a blue 2D person), drag it to (0,0), and check the **Show in 3D scene** checkbox in its properties. The animations of building and bomber are now at the same location, one on top of the other.
20. Open the editor of **Bomber**, delete the default animation and replace it with the **Airliner** 3D object. Rotate the airliner so it is oriented rightwards and set its scale to 50%.
21. On the **Agent** page of the **Bomber** properties select the **Continuous 3D space type**, deselect the **Environment defines initial location** checkbox, and set the initial coordinates to (0,0,50). The bomber will therefore fly into the model space at 5km altitude.

▶ **Set the model time units and specify the velocity of the aircraft:**

22. Select the model (topmost) item in the **Projects** tree and choose **minutes** as the model **Time units** in the **Properties** view.
23. Return to the editor of **Bomber**. On the **Agent** page of its properties set the **Velocity** of the bomber to 100.

With the 10 pixels per kilometer spatial scale we assumed earlier, the bomber will cover 100 pixels = 10 km in one minute (time unit). This gives us 600 km per hour.

▶ **Create the initial version of the bomber behavior:**

24. In the editor of **Bomber**, create a parameter **target** of type **Building** (select **Other** and type **Building**). This will be the target building in the bomber mission.
25. Create the bomber statechart, as shown in Figure 3.27.

The bombers will be parameterized with the target buildings at creation time, fly directly there, gradually lowering the altitude from 5 to 1.8 kilometers, and then return to the initial point where they will delete themselves from the model. In the **On enter** field of both states we call the **moveTo()** function of the agent to initiate movement, and then the transition from the state is triggered by the agent arrival. This pattern is very common in agent based models.

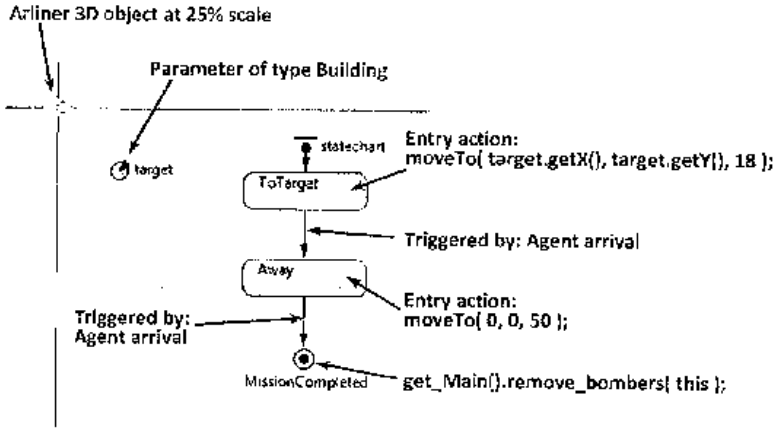


Figure 3.27 The statechart of the Bomber

▶ **Program mission assignment:**

26. Open the editor of **Main**. Add an **Event** object (from the **General** palette). Set the event properties, as shown in Figure 3.28.
27. Run the model.

The cyclic event periodically looks for a building without a bomber already flying to it. We use iteration across the agent population twice: in the outer loop we iterate across buildings, and in the inner loop we iterate across bombers. If such a building is found, we create a new bomber agent and assign the building to the bomber as the target (as long as the **Bomber** agent has one parameter **target** of type **Building**, AnyLogic generates a constructor with that parameter).

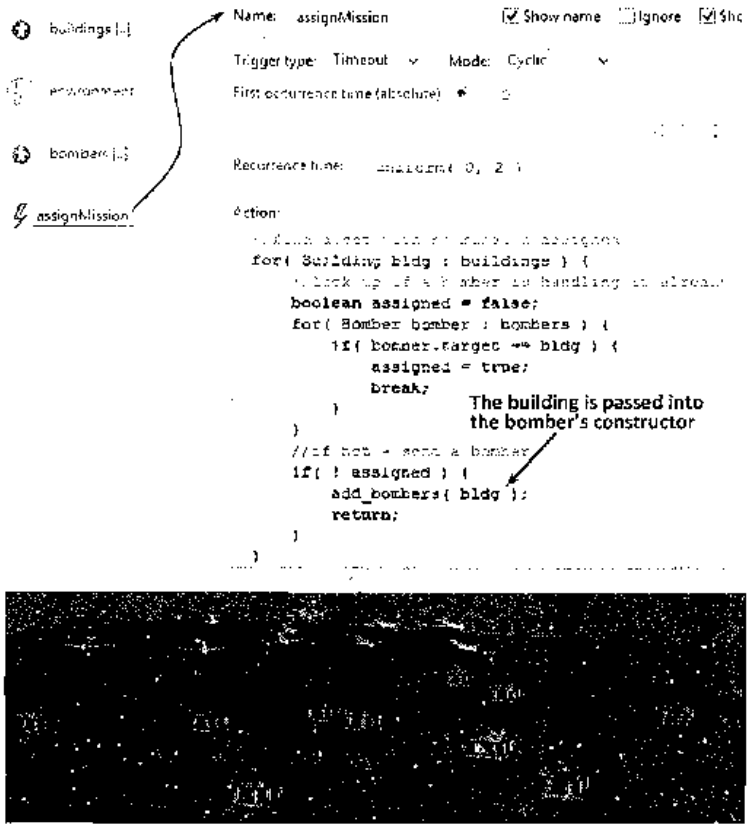


Figure 3.28 Mission assignment implemented using cyclic event

Upon creation, a bomber takes direction to the target building, makes an "instant u-turn", and heads back to (0,0). So far, the bombers use straight line trajectories – this is assumed by the **moveTo(x, y, z)** method. We will now draw the 3D "escape trajectory" for the return route and set the bombers to follow that trajectory on the way back. We will use another version of the method: **moveTo(x, y, z, polyline)**.

▶ **Draw the 3D escape route and let the bombers follow in on the way back:**

28. Open the editor of **Main** and draw a polyline, as shown in Figure 3.29. This time you can use the polyline object from the **3D** palette and not from the **Presentation** palette, because it has the **Show in 3D scene** property already selected.

29. Set the name of the polyline to **escapeRoute**, and the **Line width** to 2 pixels. On the **Advanced** page set **Z** to 20 (this will be the base Z-coordinate of the polyline) and **Z-height** to 2 pixels.

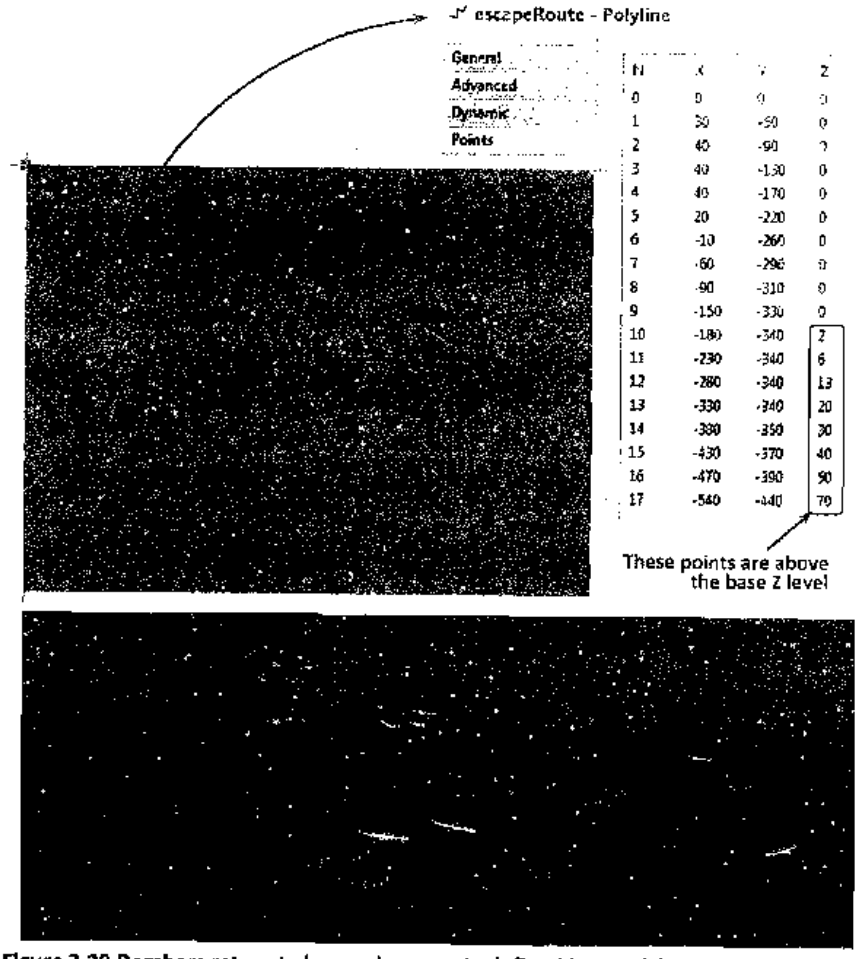


Figure 3.29 Bombers return to base using a route defined by a polyline

30. Open the **Points** page of the polyline and modify the individual Z coordinates of the points approximately, as shown in Figure 3.29. The idea is to have the initial section of the polyline at about the same altitude as the bomber attack altitude (20 pixels = 2 km is the base Z), and then gradually raise it to 9 km (base 20 + 70, which is Z of the last point).

31. Open the editor of Bomber and change the Entry action of the state Away to: moveTo(0, 0, 90, get_Main().escapeRoute); (We need to put the prefix get_Main() before the escapeRoute because this graphical object is located not inside the Bomber agent, but one level up, in Main.)
32. Run the model. See how the bombers return to the base. (The escapeRoute polyline is visible at runtime. If you would like to hide it, you can deselect the checkbox Show at runtime in its properties.)

Note that when you ask an agent to move from the point A (its current position) to the point B using a polyline, both points A and B do not need to be located on the polyline. The agent will calculate and use the shortest straight route to and from the polyline.

The next step is to model the interaction between the bomber and the target building. We will use yet another agent type, Bomb. When approaching the attack distance, the bomber will drop a bomb onto the building. When the bomb reaches the building, the building will change its state to Destroyed.

Although aircrafts carry bombs, in the model the Bomb agents will not be located inside the Bomber agent, but at the same level as the bombers and the buildings – directly inside the Main object. With this model architecture it will be easier to place bombs in the same space.

▶ **Create the Bomb agent class and program its interaction with the building:**

33. Return to the editor of Main and create another Agent population. In the wizard, make these settings:
    Agent class name: Bomb
    Population name: bombs
    Initial population size: 0
    Environment: Use existing (environment)
    Press Finish.
34. In the editor of Main, select the default animation of the bomb, drag it to (0,0) where all other agent animations are already located, and check the Show in 3D scene checkbox in its properties.
35. Open the editor of Bomb, delete the default animation and replace it with the Oval dragged from the 3D palette (this will actually be a cylinder). Place the oval at exactly (0,0), set its Fill color to black, Line color to No line, radius to 2 pixels, and Z-height to 5.

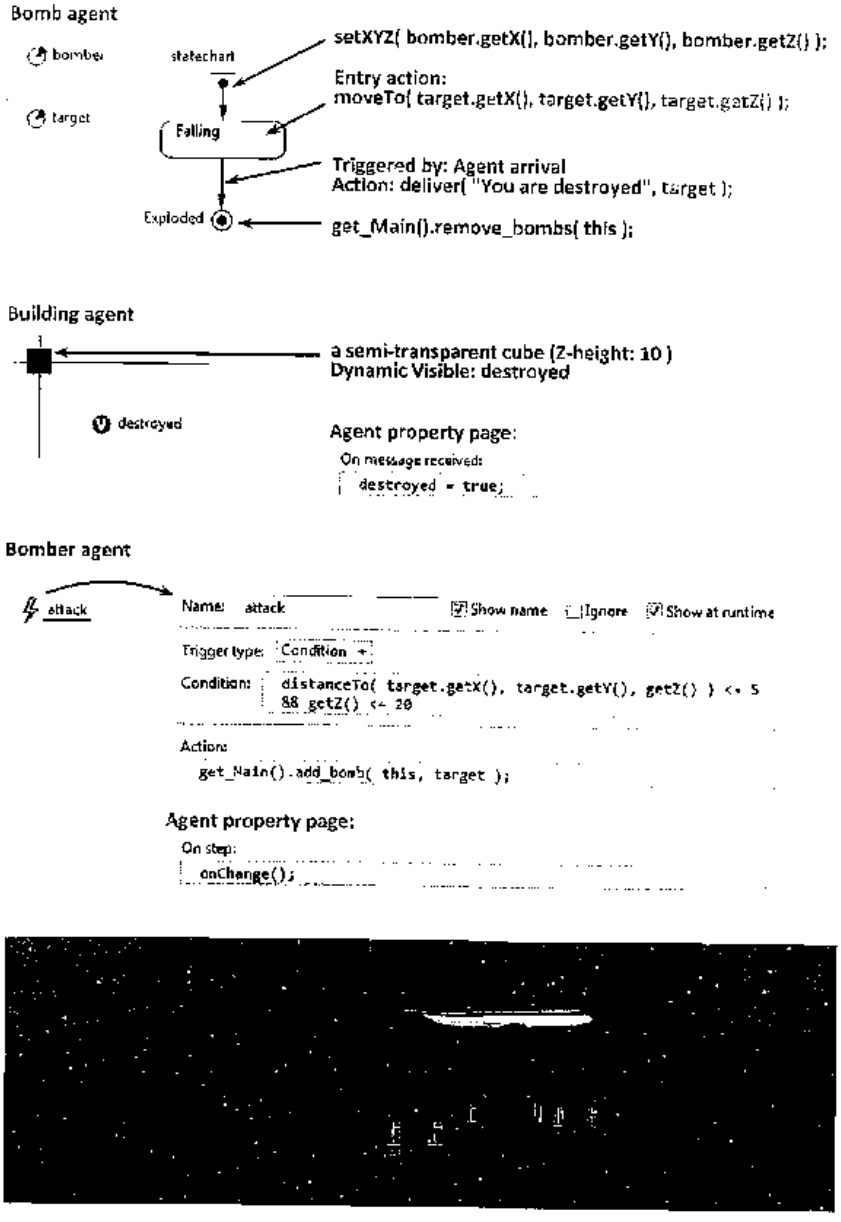


Figure 3.30 The interaction between the bomber, the bomb, and the target building

36. In the **Agent** page of the **Bomb** properties, select the **Continuous 3D space type.**

37. Create two parameters of the **Bomb** agent: **bomber** of type **Bomber** and **target** of type **Building**.
38. Draw the statechart of the **Bomb** as shown in Figure 3.30.

As you can see, the bomb sends the message "You are destroyed" to the building when it reaches the building, and then immediately destroys itself.

We are using the method **deliver()**, which delivers the message instantly (within the same event) instead of the method **send()**, which does it in a separate event (still at the model time though). This is because the message sent by the method **send()** will be discarded once the sender agent ceases to exist.

Now we will implement the building's reaction on the bomb explosion. We will add a Boolean flag **destroyed** and add some animation.

39. Open the editor of **Building** and add a Boolean variable **destroyed** with initial value **false**.
40. Open the **Agent** page of the **Building** properties and type this code in the **On message received** field: **destroyed = true;** We are not analyzing the content of the message because, for now, the only message the building may receive is the message from a bomb.
41. Create a cube around the house animation of semi-transparent red color (use the **Rectangle** shape from the **3D** palette). On the **Dynamic** page of the cube properties type **destroyed** in the field **Visible**. This way the red cube will appear around the house once the house is bombed.

The missing piece is the bomber's decision to drop a bomb. We could do it exactly upon arrival to the point directly above the target building (in the action of the transition from **ToTarget** and **Away** states). However, according to our problem definition, the bomber can do it once it gets within 500 m ground distance from the target and, at most, at 2 km altitude. To detect the moment when these conditions hold, we will introduce time steps in our model and let the bomber evaluate these conditions on each time step. The time will now be a *mixture of synchronous ticks and asynchronous events.*

▶ **Add time steps and make the bomber sense the attack conditions:**

42. Open the editor of **Main**, select the **environment** object, and check the **Enable steps** checkbox in its properties. In the **Step duration** field, type **second()**.

The default step size is one time unit (one minute in our case). However, given the velocity of the aircraft is as high as 600 km/h, under the default time step setting, the bomber will be checking the attack conditions every 10 km, which is unacceptable if we want to detect reaching 500 m ground distance from the

building. Therefore, we set the time step to one second so that the bomber will perform the checks every 167 m (10 km divided by 60). Alternatively, to using the function second(), we could write 1./60.

43. Open the editor of Bomber and add a condition-triggered event as shown in Figure 3.30.

Notice that, when creating a new bomb, the bomber passes itself (this) and the target building to the bomb constructor parameters. Another thing to notice is the calculation of the ground distance, i.e., the distance of the projections of the two objects on the ground. The bomber uses the function distanceTo(), but provides its own Z-coordinate instead of the building Z-coordinate.

44. In the Agent page of the Bomber properties, find the On step field and type onChange();
45. Run the model. Use the 3D view and zoom in to watch how the bombs are dropped and destroy the buildings.

You probably have noticed that, although all buildings are destroyed by the very first bomber attack, the new bombers are, nevertheless, sent to the target area. This happens because the mission assignment is done regardless of the state of the target assets. We will now fix it.

46. Open the editor of Main and modify the Action code of the assignMission event:

```
for( Building bldg : buildings ) {
  //destroyed buildings are ignored
  if( bldg.destroyed )
    continue;
  //look up if a bomber is handling it already
  ...
```

Now the bombers are not sent to bomb the already destroyed buildings.

The last phase of the model development is to add radars and radar-guided missiles. Both will be agents in our model. Similar to the bomber aircraft, the radar will scan the air within its coverage zone on each time step (every second). Once it detects a bomber and is able to guide yet another missile, it will launch one. The missile, similar to the bomb, will engage with the bomber (the radar will pass the bomber to the missile constructor), and explode once it gets as close as 300 m to the bomber.

As you can see, we are extensively using discrete time ("clock ticks") in this model to detect when certain geometric conditions defined over moving objects become true. Alternative approach would be to analytically calculate the exact moments of time when a bomber gets close enough to the target to drop a bomb, enters the

radar coverage zone, or when a missile catches up with the bomber. Given linear or piecewise-linear trajectories and constant velocities of bombers, this would be a task of medium (high school level) mathematical complexity and you can do it as an exercise. The resulting model will be more accurate and the simulation will be much more efficient. However, the approach we chose (recalculation of the geometric conditions on each time step) is simpler, no analytical skills are required at all, and it is also more general. It will work regardless of the type of movement.

▶ **Create the Radar agent class:**

47. In the **Projects** tree right-click the model (topmost) item and choose **New active object class** from the context menu. Give the new class the name **Radar**.
48. The editor of **Radar** opens and its properties are displayed. In the properties window check the **Agent** checkbox to enable the agent functionality.
49. Open the **Agent** page of the **Radar** properties. Choose **Continuous 3D** space type. Uncheck the **Environment defines initial location** checkbox.
50. Choose the animation shape for the radar. Among the standard 3D objects available in AnyLogic 6.9, you can use, for example, the **Truck** object (let it be the mobile air defense system). Drag it to the coordinate origin of the **Radar** editor and reduce the scale to 25%.
51. Add two parameters to the **Radar** class: **x** and **y**, both of type double. These will be the coordinates of the radar.
52. Open again the **Agent** page of the radar properties and specify the initial coordinates of the radar: (x,y,0).

Notice that this time we did not use the **Agent population** wizard to create the **Radar** class but used the **New active object class** command instead. This is because we do not plan to have a population of radars. We will create two individual radars in the model and specify individual parameters to each of them.

▶ **Create two radars in the Main class:**

53. Open the editor of **Main** and drag the **Radar** item from the **Projects** tree there. An instance of the **Radar** agent class is created in **Main** and its presentation is placed at the (0,0) point. Name this radar **radar1**.
54. In the properties of **radar1** type **environment** in the **Environment** field. This adds the **radar1** agent into the same environment where bombers, bombs, and buildings already are.
55. Ctrl+drag the **radar1** agent to create its copy. The name of the copy will be **radar2**, which is OK for us. In the properties of **radar2** press the **Create presentation** button.

56. Specify the coordinates of the radars by providing the values for their **x** and **y** parameters. Place **radar1** at, for example, (300, 350) and **radar2** at (350,200).
57. Run the model and make sure the radars are located on the bomber's route.

Now we will create the **Missile** class. The missile will be very similar to the bomb. It will have a limited lifetime from launch to explosion and will be parameterized by the target aircraft and the guiding radar. Unlike the bomb, however, the missile will periodically be adjusting its trajectory to catch up the moving target.

▶ **Create the Missile agent class:**

58. In the editor of **Main**, create yet another **Agent population**. In the wizard, make these settings:
    Agent class name: Missile
    Population name: missiles
    Initial population size: 0
    Environment: Use existing (environment)
    Press **Finish**.
59. In the editor of **Main**, select the default animation of the missile, drag it to (0,0) where all other agent animations are already located, and check the **Show in 3D scene** checkbox in its properties.
60. Open the editor of **Missile**, delete the default animation, and replace it with the **Oval** dragged from the **3D** palette (just like in the bomb, this will be a cylinder). Place the oval at exactly (0,0), set its **Fill color** to **royalBlue**, **Line color** to **No line**, radius to 1 pixel, and **Z-height** to 20.
61. In the editor of **Missile**, right-click the oval (it is now a small object at the coordinate origin) and choose **Grouping | Create a group** from the context menu. This creates a new group and adds the oval to it. In the **Dynamic** property page of the group, type **PI/2** in the **Rotation Y** field.

The reason for placing the cylinder in the group is that, by default, the cylinder stands on the (X,Y) plane and "grows" upwards along the Z axis. When the agent moves, however, its animation moves to the right along the X axis. Therefore, we need to rotate the missile body around the Y axis so that it is oriented towards the movement direction. The group (unlike individual 3D shapes) can be rotated around all three axes, and therefore is the universal method of changing the orientation of 3D shapes at runtime.

62. On the **Agent** page of the **Missile** properties, select the **Continuous 3D space type**.
63. On the same page, set the **Velocity** of the missile to 150. Remember that the model time units are minutes and the chosen space scale is 10 pixels per

kilometer. The missile therefore will cover 150/10 = 15 kilometers in one minute, which gives us the desired 900 km/h.

64. Create two parameters of the **Missile** agent: **radar** of type **Radar** and **target** of type **Bomber**.
65. Draw the statechart of the **Missile**, as shown in Figure 3.31.

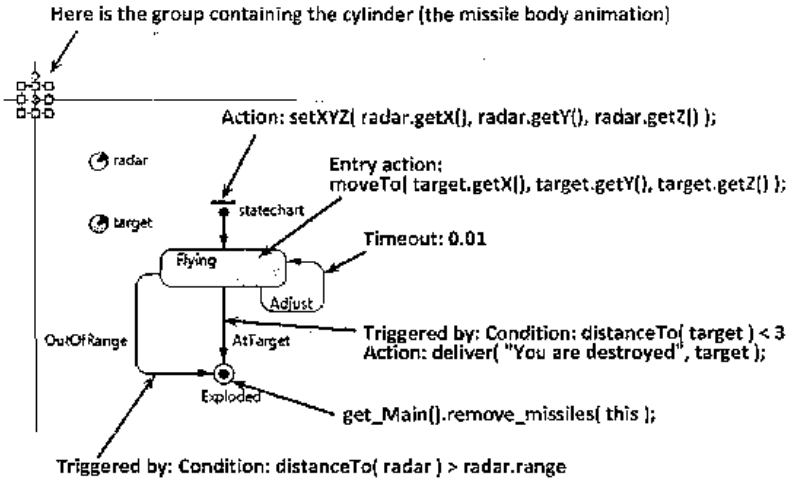


Figure 3.31 The internals of the Missile object

It makes sense to discuss the missile trajectory adjustment. The transition **Adjust** executes periodically every 0.01 minute. It has no action, but it makes the missile statechart re-enter the **Flying** state. Therefore, the entry action of the **Flying** state also executes every 0.01 minutes and makes the missile head the current position of the bomber. This way of navigation is, of course, far from ideal, but it will give us a nice curved trajectory of the missile. The condition of the **AtTarget** transition will also be re-evaluated each 0.01 minutes, or every 150 meters, which is good enough to detect the explosion moment. The same is true for the transition **OutOfRange**, which is responsible for checking if the missile leaves the radar coverage area and can no longer be guided.

▶ **Program the missile launch and guiding:**

66. Open the editor of **Radar** and add a parameter **Range** of type **double** and default value 65. This is the radius of the radar coverage zone (65 pixels equals 6.5 km).
67. Create a **Collection** inside the Radar. Name it **guidedmissiles** and set the element type to **Missile**. This collection will contain the missiles currently guided by the radar.

68. Open the **Agent** page of the **Radar** properties and type this code in the **On step** field:

```
for( Bomber b : get_Main().bombers ) { //for all bombers in the air
  if( guidedmissiles.size() >= 2 ) //if can't have more engagements, do nothing
    break;
  if( distanceTo( b ) < range ) { //if within engagement range
    //already engaged by another missile?
    boolean engaged = false;
    for( Missile m : get_Main().missiles ) {
      if( m.target == b ) {
        engaged = true;
        break;
      }
    }
    if( engaged )
      continue; //proceed to the next bomber
    //engage (create a new missile)
    Missile m = get_Main().add_missiles( this, b );
    guidedmissiles.add( m ); //register guided missile
  }
}
```

Scanning of the zone is done on every time step, i.e., every second (remember that the step size is defined in the **environment** object). The radar iterates across all bombers in the model and then across all missiles in the air. If a bomber without a missile is found, a new missile is launched.

69. Open the editor of **Missile** and add one more line to the **Action** of the **Exploded** final state:

    ```
    radar.guidedmissiles.remove( this ); //unregister with the radar
    get_Main().remove_missiles( this ); //delete self
    ```

    We need to let the radar know that the missile has reached the aircraft and exploded so that the radar can launch and guide another missile.
70. Run the model. In the 3D view see how the missiles are launched, catch up with the bombers, and disappear.

So far the missiles do no harm to the bombers because we have not programmed the bomber's reaction on the missile explosion. This will be the last bit of our model.

▶ **Model the bomber destruction:**

71. Modify the statechart of the Bomber agent, as shown in Figure 3.32. Draw a composite state around the **ToTarget** and **Away** states, and then add a transition triggered by the message **"You are destroyed"** leading to another final state.
72. Run the model.

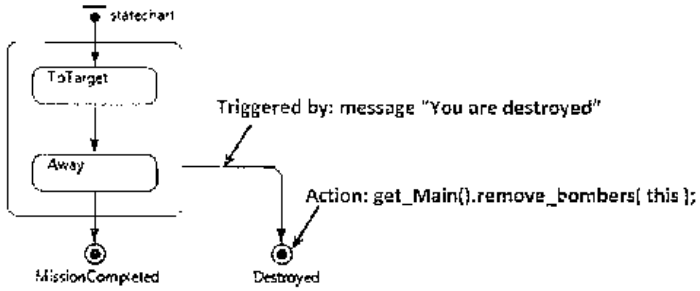


Figure 3.32 The modified statechart of the Bomber

Depending on the layout drawn, some bombers may be intercepted, and some may be able to complete their mission. You can perform various interesting experiments with this model. For example, you can optimize the location of radars, determine the time interval between the bombers, (which secures at least some of them), and so on.

On the animation side, you can visualize the radar coverage area. For that, you will need a 3D sphere object of semitransparent color, which you can add to the animation or radar and scale dynamically according to the current value of the parameter range.

Example 3.5: Agent leaving a movement trail

In the models where agents represent moving physical objects, we sometimes need to see the trajectory of the agent movement. In this example, we will create a very simple model with moving agents using the standard agent based template and then add a movement trail to the agents.

▶ **Create an agent based model with agents moving randomly:**

1. Press the **New** button on the toolbar. In the **New model** wizard, enter the model name, and on the next page choose **Use template to create model** option and **Agent Based** model template. Press **Next**.
2. In the next page of the wizard (**Setup agent properties**), set the **Initial number of agents** to 3 and press **Next**.
3. In the next page of the wizard (**Configure space and animation properties**), leave the default settings and press **Next**.
4. In the next page of the wizard (**Configure network properties**), check the checkbox **Add random movement** and press **Finish**. A new agent based model is created and the editor of its **Main** object opens.
5. Run the model. Three agents are moving randomly within the space 500x500 pixels.

In the model created by the wizard, each agent chooses a random point in the 500x500 space and starts moving there – see the **Startup code** of **Person**. Upon arrival, an agent chooses another random point and continues to the new target, this is defined in the **On arrival** field, in the **Agent** page of the **Person** properties.

▶ **Create a polyline that will be used to show the agent trail:**

6. In the **Projects** tree double-click the agent class **Person** to open its editor.
7. Open the **Presentation** palette, drag the **Polyline** object and drop it anywhere in the editor of **Person**.
8. In the **General** page of the polyline properties set the name to **trail**.
9. In the **Dynamic** page of the polyline properties set:
   X: -getX()
   Y: -getY()
10. Right-click the polyline and choose **Grouping | Create a group** from the context menu. A new group is created; it contains the polyline **trail**.
11. In the **Dynamic** page of the group properties set **Rotation to: –getRotation()** and enter the following code in the **On draw** field in the same page:
    trail.setPoint(trail.getNPoints()-1, getX(), getY());
12. Move the group to the coordinate origin in the editor of **Person**.

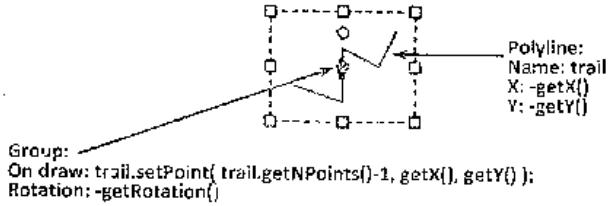


Figure 3.33 Trail of the agent is implemented as a polyline in a group in the agent animation

We need the trail of the agent to be fixed in the space where the agents move. The polyline, however, belongs to the presentation of the agent, therefore it will move with the agent. To compensate the movement, we set the coordinates of the polyline opposite to the coordinates of the agent. Similarly, we need to compensate the rotation of the agent. This is done by adding the polyline to a group and rotating the group in the direction opposite to the agent. Finally, we need to continuously extend the last segment of the trail to the current position of the agent. Therefore, whenever the group with the polyline is drawn, we set the coordinates of the last point of trail to the current position of the agent.

▶ **Set the initial shape of the trail and add a new segment on each arrival**

13. In the **Startup code** field of the **General** page of the **Person** properties, enter the following code *before* the auto-generated call of **moveTo**:

```
//leave two points in the polyline
trail.setNPoints( 2 );
//place both at the current agent position
trail.setPoint( 0, getX(), getY() );
trail.setPoint( 1, getX(), getY() );
//the call of moveTo generated by the wizard follows
```

14. In the **Agent** page of the **Person** properties, enter the following code in the **On arrival** field *before* the auto-generated call of **moveTo**:

```
int n = trail.getNPoints();
trail.setPoint( n-1, getX(), getY() ); //set the last point to the current position
trail.setNPoints( n+1 ); //add a new point
trail.setPoint( n, getX(), getY() ); //place it at the same position for now
//the call of moveTo generated by the wizard follows
```

15. Run the model. The three agents are leaving their trails as they move. You can apply the virtual time mode for a short time to let the agents move long distances.

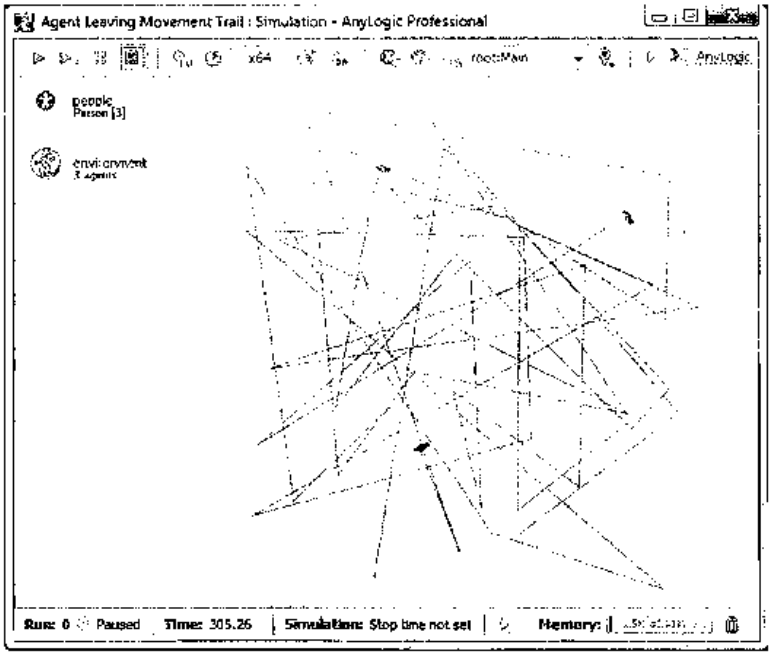


Figure 3.34 The three agents leave their trails

When the model starts, the trail should have two points (one segment): the start point at the initial position of the agent and the end point that will move as the agent moves, initially at the same place. Upon arrival, we need to close the current segment of the trail and start a new one. This is done in the **On arrival** code.

Finally, we will add a text displaying the distance traveled by the agent.

▶ **Add text displaying the distance traveled**

16. Open the editor of **Person**. Open the **Presentation** palette, drag the **Text** object, and drop it nearby the person shape.
17. In the **Dynamic** page of the text properties type **trail.length()** in the **Text** field.
18. Right-click the text shape and choose **Grouping | Add to existing group** from the context menu. Click on the highlighted group, which you have previously placed at the coordinate origin.
19. Run the model.

The length of the **trail** equals the distance traveled by the agent, so we can simply use the method **length()** of the polyline. By adding the text shape to the group, we prevent the text from rotating with the agent.

? The length of the trail implemented that way is unlimited, and after a long period of simulation the picture may become less informative and may slow down the animation. How would you modify the model so that the trail will contain the maximum 50 most recent segments? How would you keep the information on the distance traveled correct?

Continuous space API

The following methods, defined in the class **AgentContinuous**, are common for agents living in 2D, 3D, and GIS continuous spaces:

- **double distanceTo(Agent other)** – calculates the distance from this agent to another agent.
- **AgentContinuous getNearestAgent(java.lang.Iterable<? extends AgentContinuous> agents)** – returns the nearest agent from the given collection.
- **double getRotation()** – returns the current rotation angle (in radians) of the agent in XY plane.
- **double getTargetX()** – returns the X coordinate of the target location when moving, otherwise the current X coordinate of the agent.
- **double getTargetY()** – returns the Y coordinate of the target location when moving, otherwise the current Y coordinate of the agent..
- **double getVelocity()** – returns the velocity of the agent regardless of whether the agent is moving or not.
- **double getX()** – returns the current (up-to-date) X coordinate of the agent.
- **double getY()** – returns the current (up-to-date) Y coordinate of the agent.
- **boolean isMoving()** – tests whether the agent is currently moving.
- **void moveToNearestAgent(java.lang.Iterable<? extends AgentContinuous> agents)** – starts movement towards the nearest agent from the given collection.
- **void setVelocity(double v)** – changes the velocity of the agent (in 2D and 3D space the velocity is measured in pixels per model time unit, and in GIS-based environments – in m/s).
- **void stop()** – stops movement.
- **double timeToArrival()** – returns the time to arrival to the target location if the agent is moving or 0.

The following functions are specific to 2D space; they are defined in the class **AgentContinuous2D**:

- **double distanceTo(double x, double y)** – calculates the distance from this agent to a given point.
- **void jumpTo(double x, double y)** – instantly places the agent at a given location.

- **void moveTo(double x, double y)** – starts movement towards the given target location.
- **void moveTo(double x, double y, Path2D path)** – starts movement towards the given target location along a given path (polyline or line).
- **void setXY(double x, double y)** – sets the coordinates of the agent location; should be used for initial setup only; assumes the agent is not moving.

These are the functions of **Agent Continuous3D** specific to 3D space:

- **double distanceTo(double x, double y, double z)** – calculates the distance from this agent to a given point.
- **double getTargetZ()** – returns the Z coordinate of the target location when moving, otherwise the current Z coordinate.
- **double getVerticalRotation()** – returns the current vertical rotation angle of the agent.
- **double getZ()** – returns the current (up-to-date) Z coordinate of the agent.
- **void jumpTo(double x, double y, double z)** – instantly places the agent into a given location.
- **void moveTo(double x, double y, double z)** – starts movement towards the given target location.
- **void moveTo(double x, double y, double z, Path3D path)** – starts movement towards the given target location along a given path.
- **void setXYZ(double x, double y, double z)** – sets the coordinates of the agent location.

3.7. Networks and links

In many (but not all) agent based models, agents have relationships with each other. If agents are people, the relationships may be friendship, kinship, sexual relationships, relationships at work, or physical contacts. If agents are IT infrastructure objects, they may be physical or wireless links. In a supply chain model, they may be producer-consumer relationships, and so on.

AnyLogic offers several ways to model agent relationships. They are:

- Standard and user-defined networks with uniform bidirectional links (friendship, common interest, information exchange)
- Custom unidirectional links with possibly different meanings (child-parent, boss-subordinate, client-salesman, workstation-server, producer-consumer)
- Hierarchical model architectures (company-employee, town-habitant, group-member)

- Using ports to connect agents

Do not think of links and networks as things that are set up once and are not able to change. As the model is running, you can modify the relationships between the existing objects, add new objects and establish their connections, delete objects, restore the standard network types, and so on.

Standard networks

AnyLogic supports several standard types of networks and you are able to modify them and create your own networks. The standard networks are based on *uniform bi-directional connections* between agents. The standard network types are (see Figure 3.35):

- *Random.* An agent is connected (on average) to a given number of randomly selected other agents.
- *Distance-based.* Two agents are connected if the distance between them is within a given range. This network type applies only to agents in continuous space.
- *Ring lattice.* Regardless of the space and layout type, the agents are considered to be evenly distributed on a virtual ring. An agent is connected to a given number of agents closest to it on the ring.
- *Small world.* Same virtual ring is assumed. An agent is connected to a given number of agents, most of which are its closest neighbors, but a certain percentage of connections are long-distant. This network is constructed from a pure ring lattice by re-wiring some connections to long-distant. Examples of small world networks are social networks or Internet connectivity network. An interesting property of a small world network is that the number of hops between two randomly chosen agents is proportional to the logarithm of the number of agents, which means even strangers are linked via mutual acquaintances ("Small-world network", (n.d.)).
- *Scale free.* Formally, this is a network where the distribution of number of agent's connections follows a power law, i.e., the fraction of agents with k connections is $k^{-\gamma}$, γ being the network parameter. In such networks, some agents are "hubs" with a lot of connections, and some are "hermits" with few connections. Some real world networks are claimed to be scale-free, such as professional collaboration networks, computer networks, or airline networks ("Scale-free network", (n.d.)).

A good visual demonstration of the standard network types is available in the example model "Agent Network and Layout Demo" included in the AnyLogic How-to example set.

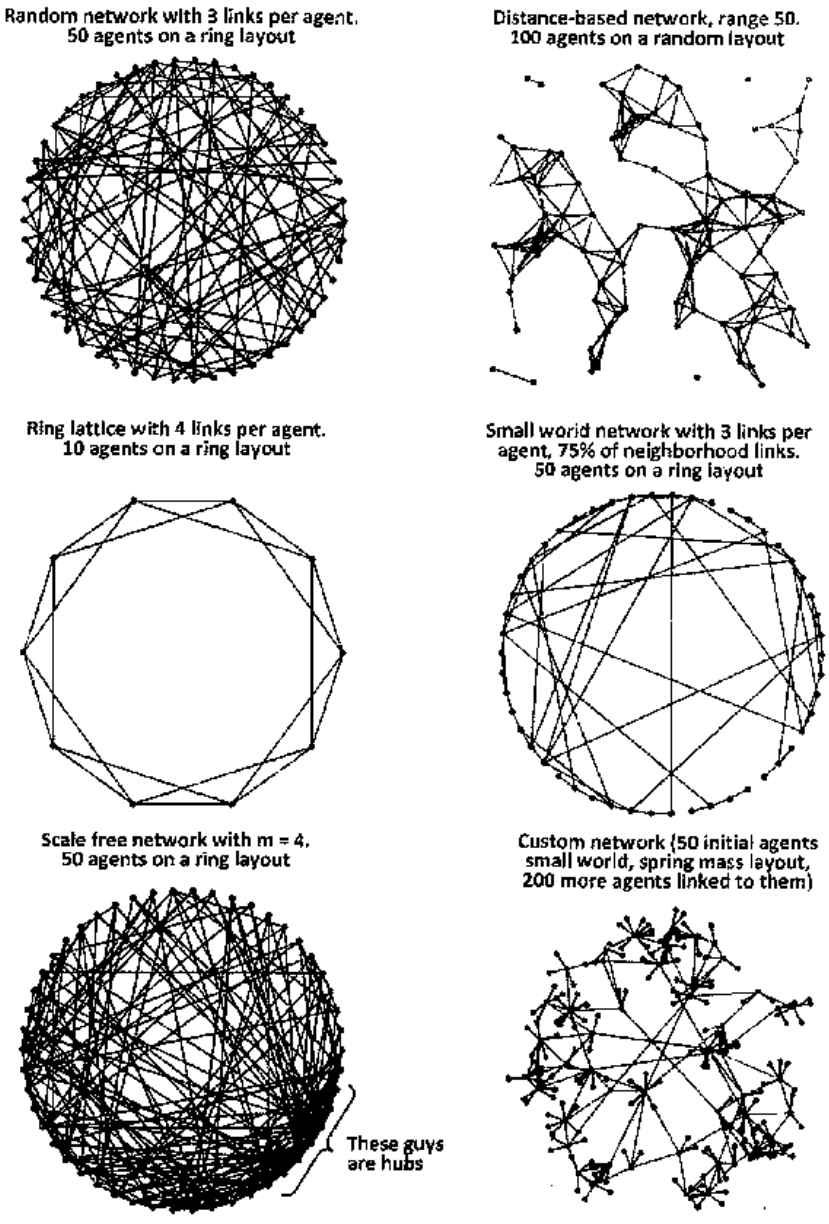

Random network with 3 links per agent.
50 agents on a ring layout
Distance-based network, range 50.
100 agents on a random layout
Ring lattice with 4 links per agent.
10 agents on a ring layout
Small world network with 3 links per
agent, 75% of neighborhood links.
50 agents on a ring layout
Scale free network with m = 4.
50 agents on a ring layout
Custom network (50 initial agents
small world, spring mass layout,
200 more agents linked to them)
These guys
are hubs

Figure 3.35 Standard and custom network types

In AnyLogic, networks are constructed with the help of the **Environment** object and can connect agents of different types (different classes), given they are members of the same environment.

▶ **To create a network:**

1. Select the **Environment** object and open the **Advanced** page of its properties. Select the network type and specify the parameters of the network.

This has the following effect. The agents that are created at the model startup get connected according to the given network parameters. If you add more agents later on, or delete existing agents, the network will not automatically reconfigure itself unless you call the function **applyNetwork()** of the **Environment** object.

Example 3.6: Periodic repair of a standard network

In this example, we will create a population of agents in a rectangular 2D space and connect them based on inter-agent distances. We will let the agents die, be born, and move, and will set up a periodical "repair" of the network. We will also show how to visualize links between the agents.

▶ **Create agents connected with a distance-based network:**

1. Create a new model. Drag the **Agent population** object from the **General** palette to the editor of **Main**.
2. On the first page of the wizard, leave the default settings. On the second page, set the space size to 500 by 400 and set the **Network type** to **Distance based** with parameter 50.
3. Run the model. The agents are randomly distributed across the area and they are connected, but we do not see the links yet.

▶ **Visualize the links between agents:**

4. Open the editor of **Person** (this is the default agent class name given by the wizard). Drag a **Line** object from the **Presentation** palette and place the beginning of the line to the coordinate origin.
5. Open the **Dynamic** page of the line properties and type:
   Replication: getConnectionsNumber()
   Rotation: -getRotation()
   dX: getConnectedAgent(index).getX() - getX()
   dY: getConnectedAgent(index).getY() - getY()
6. Run the model. Now you can see the links and make sure the connections are distance-based.

This works as follows. The number of line copies is set to the number of connections the agent has. The beginning of each line is at the coordinate origin (at the center of the agent). The end of the line is placed at the center of the connected agent by

setting dX and dY to the differences of coordinates of this and the other agent. The rotation of the line is set to compensate the rotation of the agent, which moves with the right-hand side in front.

As the standard connections are bidirectional, for each connection, there will be not one, but two lines displayed one on top of the other, which is fine for our simple example. Should you need to display only one line, you can set the color of the line to null if **getIndex()** or **hashCode()** of the other agent is greater than **getIndex()** of this agent.

▶ **Make the agents move:**

7. Open the editor of **Person** and write this statement in the Startup code: **moveTo(uniform(500), uniform(400));**
8. Open the Agent page of the Person properties and write exactly the same code in the **On arrival** field. This will make the agents constantly move in the space.
9. Run the model. See how the agents move and how the links, after a while, cease to reflect the distances between the agents.

▶ **Make the agents die and be born:**

10. Open the editor of **Person** and add an **Event** object from the **General** palette. Name the event **death**, leave the **Timeout** trigger type and set the **Mode** to **User control**. Set the timeout value to **uniform(60, 100)**. In the event **Action** write: **get_Main().remove_persons(this);**. This event will not be scheduled unless we explicitly do it.
11. In the **Startup code** of **Person**, add the line: **death.restart();**. This will schedule the death of the agent within 60-100 time units from its birth.

We could not use a **Timeout** event with the mode **Occurs once** because then the occurrence time is absolute (counted from the model startup), and the agents born later than time 100 will never die.

12. Run the model. Wait for all agents to die.
13. Open the editor of **Main** and add an **Event** object with the name **birth**. Set the **Trigger type** to **Rate** with rate value 1. The action of the event should be **add_persons();**. Now, on average, every time unit a new agent will be born.
14. Run the model.

After a while the number of agents stabilized at about 80 (do you know why, by the way?), but there are no links between them because no one is taking care of connecting the new born agents. The last bit of the model is the periodic repair of

the distance-based network. We can implement this as yet another periodic event at the Main level.

▶ **Setup the periodic repair of the network:**

15. Add another event in **Main** object with the name **repairNetwork**. Make it a **Cyclic Timeout** with **Recurrence time** 5. The **Action** of the event will be: **environment.applyNetwork();**. This will rewire the agent connections according to the network type specified in the **environment** object (the network type can also be changed dynamically if needed).
16. Run the model. See how the network is repaired every 5 time units.

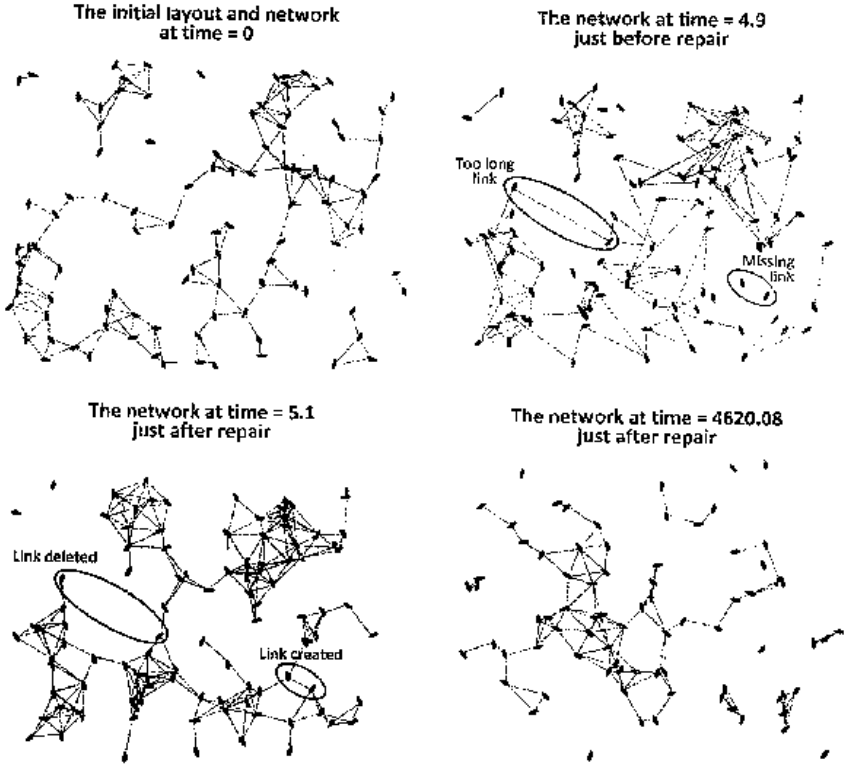

The initial layout and network at time = 0
The network at time = 4.9 just before repair
Too long link
Missing link
The network at time = 5.1 just after repair
Link deleted
Link created
The network at time = 4620.08 just after repair

Figure 3.36 Distance-based network with periodic repair

Example 3.7: Custom network built using standard connections

We will first create a standard network of 20 agents connected in a ring, and then add 60 more agents and connect 3 new agents to each initial agent. We will then apply a spring-mass layout to better visualize our network.

▶ Create 20 agents connected in a ring:

1. Create a new model. Drag the **Agent population** object from the **General** palette to the editor of **Main**.
2. On the first page of the wizard, set the **Initial population size** to 20. On the second page, set the **Network type** to **Ring lattice**. Click **Finish**.
3. Select the **environment** object in the editor of **Main** and set the **Layout type** to **Ring** on the **Advanced** page of its properties.
4. Repeat steps 4-6 of the previous model, Example 3.6: "Periodic repair of a standard network", to visualize the links.
5. Run the model. The agents are evenly distributed on a ring and linked (by default, there are two connections per agent; you can change this in the properties of the **environment**).

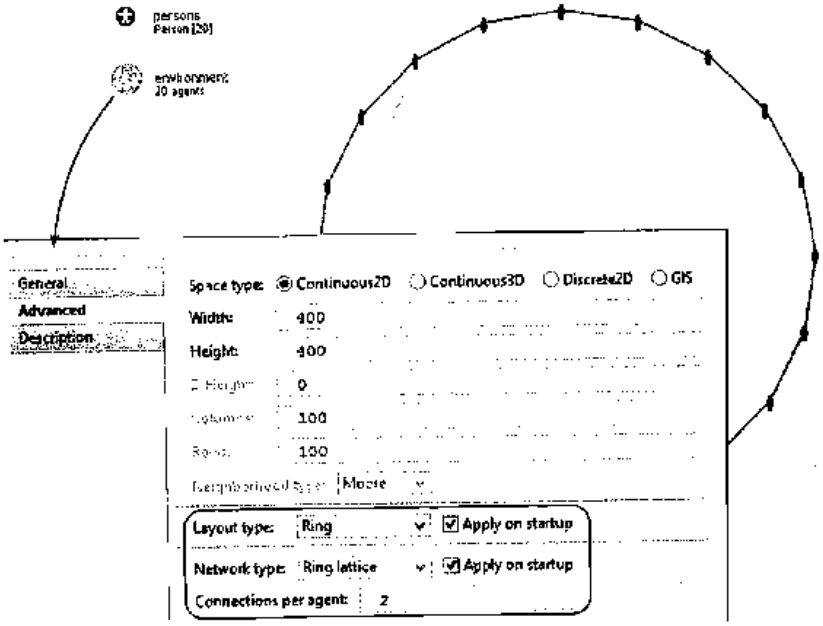


Figure 3.37 A simple ring lattice – the backbone of the future custom network

▶ **Dynamically create 100 additional agents and link them to the existing agents:**

6. Add this **Startup code** in the **Main** object:

```
for( int i=0; i<20; i++ ) { //iterate across the initial 20 agents
  Person hub = persons.get(i); //for each initial agent (a "hub")
  for( int j=0; j<3; j++ ) { //create 3 more agents
    Person sub = add_persons();
    sub.connectTo( hub ); //and link to the hub
  }
}
```

7. Run the model. The new agents are randomly spread across the area and linked to the initial 20 agents.
8. Add two more lines at the end of the **Startup code** of **Main** (note that after you have set the new layout type you need to call the function **applyLayout()** to actually relocate the agents):

```
//change the layout type
environment.setLayoutType( Environment.LAYOUT_SPRING_MASS );
//and apply the new layout
environment.applyLayout();
```

9. Run the model and see the new spring mass layout.

The key function used in this example is the agent's function **connectTo()**, which creates a new bidirectional connection between this and another agent. The standard networks are built of connections of the same type.

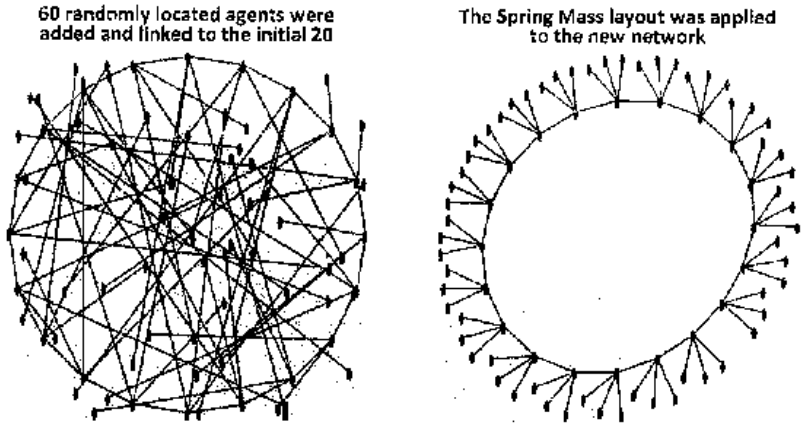


Figure 3.38 The custom network before and after applying the spring mass layout

Fully connected networks

It does not make a lot of sense to use connections to establish a *fully connected network* (one where everybody is connected to everybody else). Building and maintaining such a network may require significant computer resources, whereas you can easily use AnyLogic agent, environment, or replicated object API to iterate through the collection of agents, or to access a particular or a random agent.

For example, if all of the agents of the class Person are elements of a single replicated object (like people embedded in Main class), to access the set of agents from inside one of them, you can use the function:

- **ActiveObjectList< Person> getReplicatedList()** – returns the list you can iterate through, access a particular agent by index by calling its **get(i)** function, query statistics, or pick a random agent by calling **random()**.

If the set of agents consists of *several subsets* (for instance, you have two types of agents or two replicated objects), but all of them live in the *same environment*, you can use the API of the **Environment** object. From inside an agent you can call:

- **getEnvironment().getAgentCollection()** – returns the collection of all agents in the environment, again, with possibilities of iteration and access by index.
- **getEnvironment().getRandomAgent()** – returns a randomly chosen agent in the environment.

When other agents are accessed via the environment, the returned class will be one of the generic classes, like **AgentContinuous2D**, so you may need to cast (convert) it to the actual class to access its specific properties. Please also note that all those functions may return the calling agent, so this may need to be checked.

Network and layout-related API

The following methods are defined in the class **Agent**, and thus are common for all types of agents:

- **LinkedList<Agent> getConnections()** – returns the list of all connected agents, or null if there have been no connections established.
- **int getConnectionsNumber()** – returns the number of connected agents.
- **Agent getConnectedAgent(int index)** – returns the connected agent with the given index. Note that the method returns the generic type **Agent** and you may need to cast it to a specific type.
- **connectTo(Agent a)** – adds a given agent to the connections of this agent and vice versa.

- **boolean isConnectedTo(Agent a)** – tests if this agent is connected to a given agent.
- **boolean disconnectFrom(Agent a)** – disconnects this agent from a given other agent, returns false if the agents were not connected.
- **disconnectFromAll()** – disconnects the agent from all other agents.

In addition, the **Environment** class offers this API:

- **applyNetwork()** – discards all existing connections and establishes new connection network according to the current network settings.
- **getNetworkType()** – returns the network type, one of: **NETWORK_ALL_IN_RANGE, NETWORK_RANDOM, NETWORK_RING_LATTICE, NETWORK_SCALE_FREE, NETWORK_SMALL_WORLD, NETWORK_USER_DEFINED**; remember that, as long as the constants are defined in the class **Environment**, you should include the prefix "**Environment.**" when you use them.
- **setNetworkRandom(double connectionsPerAgent)** – sets the network type to **NETWORK_RANDOM** with a given average number of connections per agent.
- **setNetworkRingLattice(int connectionsPerAgent)** – sets the network type to **NETWORK_RING_LATTICE** lattice.
- **setNetworkScaleFree(int m)** – sets the network type to **NETWORK_SCALE_FREE**.
- **setNetworkSmallWorld(int connectionsPerAgent, double neighborLinkProbability)** – sets the network type to **NETWORK_SMALL_WORLD**.
- **setNetworkUserDefined()**– sets the network type to **NETWORK_USER_DEFINED**.

The 2D-specific class **Environment2D** additionally has these functions:

- **int getLayoutType()** – returns the current layout type, one of **LAYOUT_ARRANGED, LAYOUT_RANDOM, LAYOUT_RING, LAYOUT_SPRING_MASS, LAYOUT_USER_DEFINED**. Again, the prefix "**Environment.**" is needed when you use these constants.
- **setLayoutType(int type)** – sets the layout type.
- **applyLayout()** –relocates the agents according to the selected layout type.

Unidirectional, temporary, and other custom types of links

Uniform bidirectional connections are not always the desired way of modeling the relationships of real world objects. For example, I may listen to somebody's opinion, but that person does not necessarily know me, thus our relation is asymmetric. In many cases, connections must have specific names so that we can distinguish between them, like Husband, Wife, Kids, Colleagues, Friends, Boss, and so on. The links Wife and Husband are naturally unidirectional, while the link Spouse is bidirectional.

In AnyLogic you can establish and maintain all kinds of connections using references to agents stored in plain variables (see Section 10.3) or collections (see Section 10.6). This can be considered as an alternative or as an addition to the standard networking service provided by the **Environment** object with the functions **getConnections()**, **connectTo()**, etc. Consider Figure 3.39. Here kinship between individuals is modeled by custom links. All kinds of kinship are implemented as pairs of unidirectional links, either symmetric (like Spouse) or asymmetric (like Father and Kid).

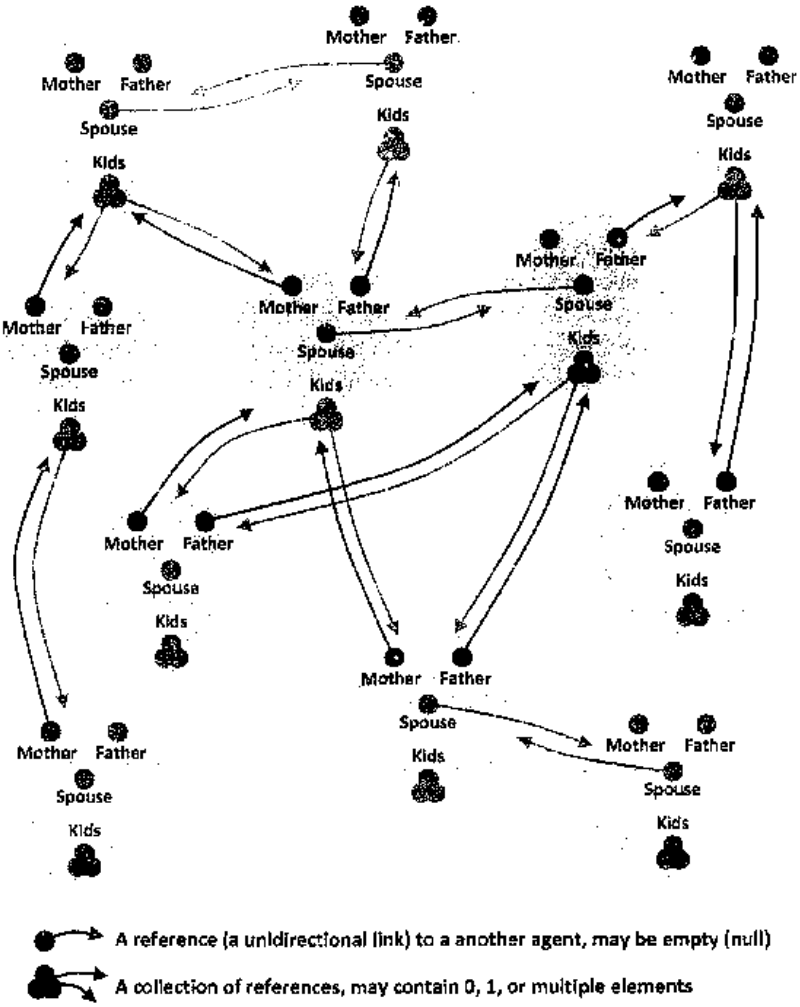

Mother
Father
Spouse
Kids
A reference (a unidirectional link) to a another agent, may be empty (null)
A collection of references, may contain 0, 1, or multiple elements

Figure 3.39 Custom links used to model kinship relations

Custom links are a very flexible mechanism for creating specific networks, but the modeler who uses them becomes responsible for maintaining the network consistency. For example, when an agent dies or is otherwise deleted from the model, all references to that agent should be cleared throughout the model.

Example 3.8: Kinship modeled using custom links

We will create a fairly simple agent based population model where people (males and females) are born, grow up, marry, have kids, get old, and die. We will maintain the kinship relations, as shown in Figure 3.39, by means of custom links (references) to other agents. We will also take care of the kinship network consistency.

▶ **Create the initial population of 300 people and create custom links:**

1. Create a new model and use the **Agent population** wizard to create 300 agents. Set the name of the population to **people**.
2. Open the editor of **Person** and replace the default animation with the circle of 2 pixel radius located at the coordinate origin. Name the circle **circle**.
3. In the **Person** class, create a Boolean parameter **male** with the initial value **randomTrue(0.5)**. The birth probability for males and females will be the same.
4. From the **General** palette, drag the **Variable** and name it **mother**. Set the type of the variable to **Person**. This will be the reference (link) to the person's mother. Agents in the initial population will not have live parents, and this variable will be null.
5. Copy the variable to create two more links: **father** and **spouse**.
6. Drag the **Collection** object and name it **kids**. Set the element type of the collection to person. This will be a collection of references to the person's children, initially empty.
7. Open the **Agent** page of the **Person** properties and deselect the checkbox **Environment defines initial location** (parents will take care of the initial location of their kids).
8. On the same page set the **Velocity** of the agent to 100. We will use movement to visualize the family formation.
9. Draw two lines with the start points at the coordinate origin of the **Person** and dynamic properties, as shown in Figure 3.40. These lines will visualize the kinship relations of the person. Note that we are only visualizing the relation wife and kid, and we do not draw the lines for husband and father/mother to avoid duplication.
10. Run the model to make sure the 300 people are in place and there are no links between them so far.

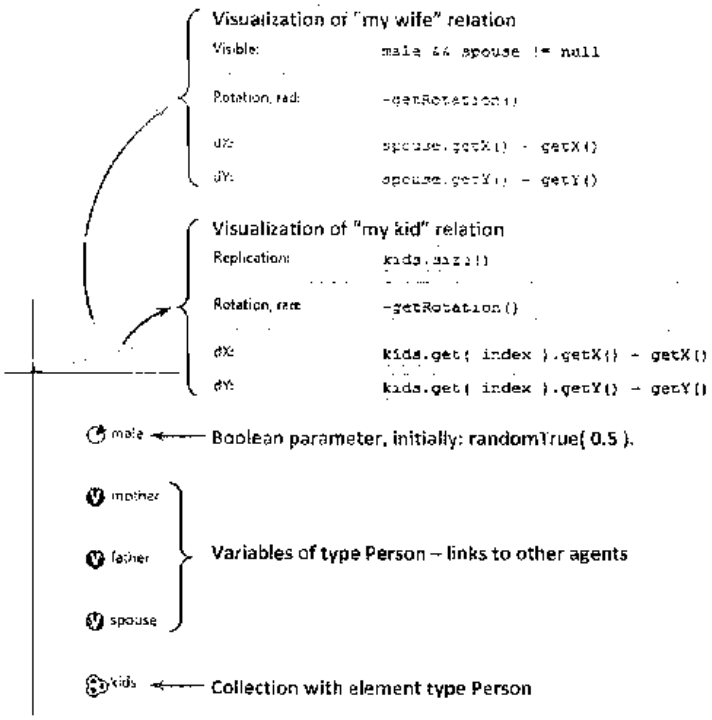


Figure 3.40 Links to the agents in kinship and their visualization

▶ **Add person's behavior:**

11. Draw the statechart, as shown in Figure 3.41.
12. Specify the triggers and actions of the transitions as explained below.
13. Run the model.

As you can see, the person may reach the **Adult** state at any age between 16 and 25. Then, for another 15-25 years the person may create a family and have kids. Then he or she becomes a **Senior** and dies after yet another 15-25 years.

Transition LookForWife and its branches Found and NoLuck. The transition has a **Guard** that is set to **male**, which means that only a male person can execute it. The transition is triggered at the **Rate** of 1, i.e., on average once a year. The **Action** of the transition is:

```
//look for wife on average once a year
for( Person p : get_Main().people ) { //search all people
  if( ! p.male && p.statechart.isStateActive( NoFamily ) ) {
    //if a person is female and not married, marry her
    spouse = p;
```

```
        send( this, p ); //and let her know
        break; //exit the loop
    }
}
```

If a single adult male finds a single adult female, he marries her, which is implemented by setting the male's spouse link and sending the wife the message containing a reference to the male himself ("this"). The condition of the branch **Found** is **spouse != null**, otherwise the branch **NoLuck** takes the male back to the **NoFamily** state.

The transition GetMarried has a **Guard** set to **!male** so that it applies to females only. It is triggered by a message of type **Person** (remember that, when getting married, a male sends a reference to himself to the chosen wife). The **Action** of the transition is:

```
spouse = msg; //remember husband and move to him
moveTo( spouse.getX() + 10, spouse.getY() );
```

This means the wife sets up the reference to the husband and moves to his location.

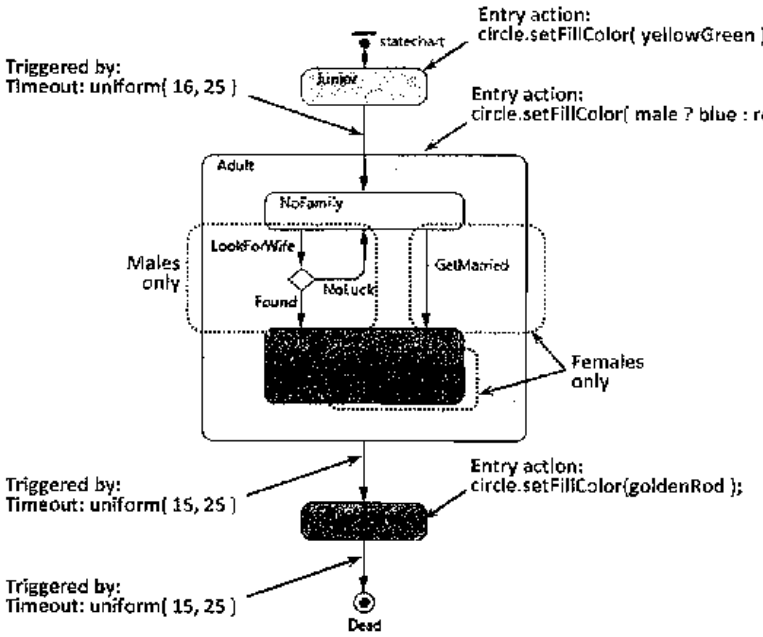


Figure 3.41 The statechart of a person. Other transitions are explained in the main text

The transition NewKid is an internal transition inside the state **Family**. It is guarded by the expression **! male && spouse != null**, so that only females with live husbands can have new kids. The transition is taken at the **Rate** of 0.1, i.e., on average once every 10 years. The action is:

```
//a kid is born on average every 10 years
Person kid = get_Main().add_people(); //new person
kids.add( kid ); //add to my kids
spouse.kids.add( kid ); //add to husband's kids
kid.mother = this; //assign mother to the kid
kid.father = spouse; //assign father
//place the kid nearby the father
kid.jumpTo( spouse.getX() + uniform( -10, 10 ), spouse.getY() + uniform( -10, 10 ) );
```

A newborn needs some "wiring" to be done. We establish the cross-links between the kid and his parents, and place the kid near his father.

And finally, the state Dead. Here we delete the person from the model, but, before he/she disappears, we must clean all references to the person to maintain the network consistency. The action of the state **Dead** is:

```
//clean up all links to myself
//from kids
for( Person kid : kids ) {
  if( male )
    kid.father = null;
  else
    kid.mother = null;
}
//from spouse
if( spouse != null)
  spouse.spouse = null;
//from parents
if( father != null )
  father.kids.remove( this );
if( mother != null )
  mother.kids.remove( this );
//and die
get_Main().remove_people( this );
```

Note that, before addressing a spouse or parent, we need to check if the other person is alive.

The model shows interesting behavior, see Figure 3.42.

? What can be changed in the model to ensure the population growth (give options)?

? In the current model, brothers and sisters can possibly marry. How would you modify the model to prevent this?

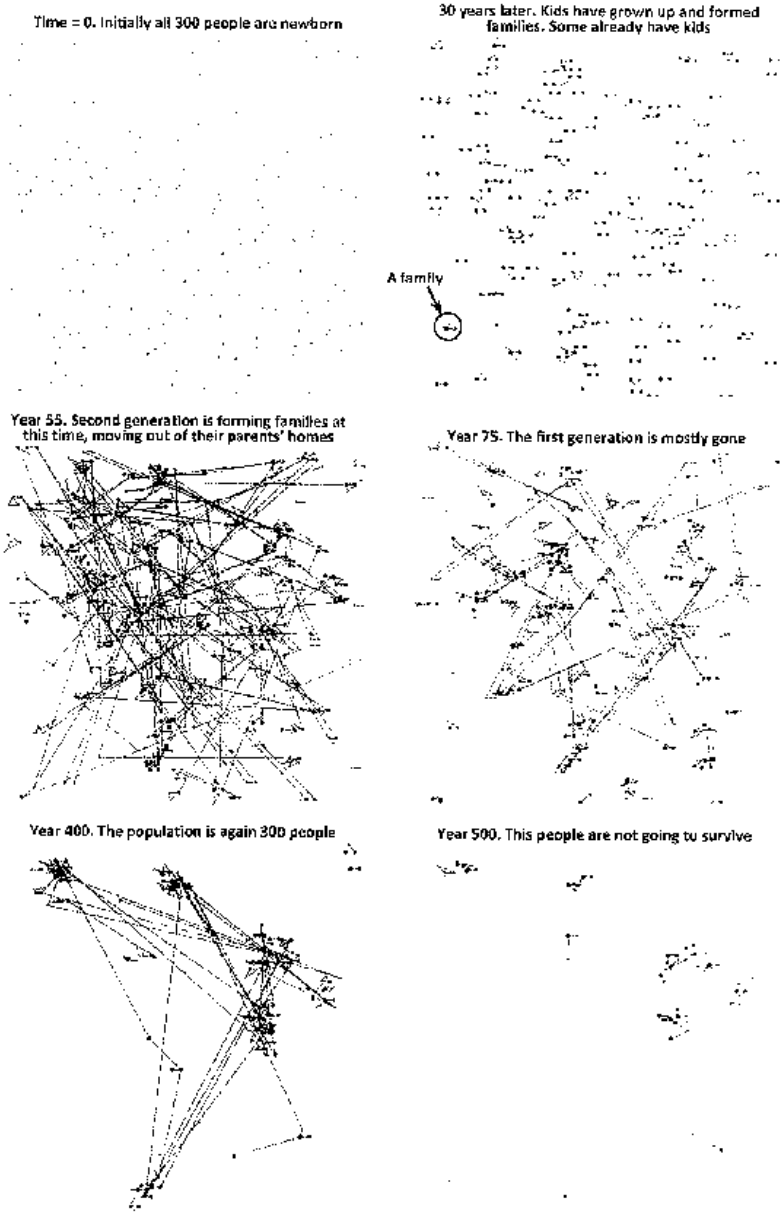

Time = 0. Initially all 300 people are newborn
30 years later. Kids have grown up and formed families. Some already have kids
A family
Year 55. Second generation is forming families at this time, moving out of their parents' homes
Year 75. The first generation is mostly gone
Year 400. The population is again 300 people
Year 500. This people are not going to survive

Figure 3.42 Behavior of the Kinship model

A note on vertical links in hierarchical models

Note that if the model is hierarchical, for example, the agent of type **Department** contains agents of type **Employee**, and is itself embedded into the agent of type **Company**, you do not have to establish special "vertical" links: AnyLogic active object hierarchy naturally provides them: an employee can always access his department by calling **get_Department()**, and the company can address any of its departments by index, e.g., **departments.get(i)**. Also see Section 10.9.

Using ports to connect agents

Sometimes you want to manually lay out the agents on the canvas and connect them at design time. This may make sense if the number of agents is not too large, and their relationships are stable and known in advance. In such cases, AnyLogic ports may serve as connection points.

Consider a model of a supply chain. Each element in the chain (a producer, a distribution center, a retailer) is an agent with two ports, see Figure 3.43. The incoming orders and outgoing shipments are received and sent by the right-hand side port. Correspondingly, the outgoing orders and incoming shipments are sent and received by the left-hand side port. Shipments and orders are AnyLogic messages, i.e., arbitrary Java objects.

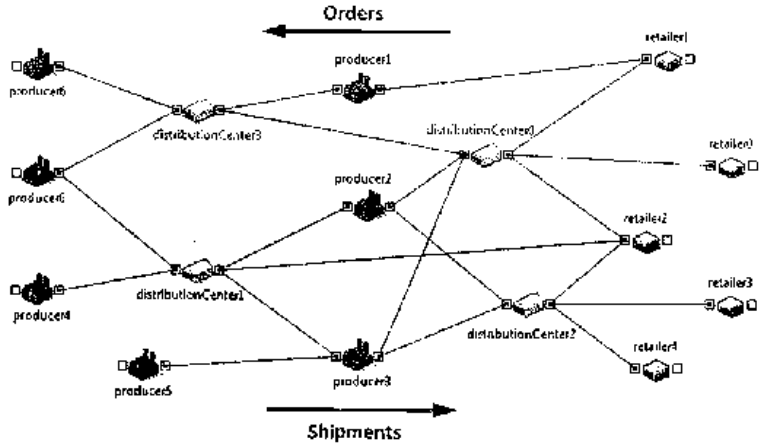


Figure 3.43 A model of a supply chain. Fixed layout, agents are connected via ports

If a port is connected to multiple other ports, you may need to include the destination object into the message and perform filtering at the receiving end so that the message gets handled only by the intended recipient.

AnyLogic ports can be connected and disconnected dynamically if needed. This leaves some flexibility to such structures.

3.8. Communication between agents. Message passing

- **You can consider an AnyLogic model as an *open space* where everybody can see and talk to everybody else.**

You can access any construct in any active object, from any other construct in any other active object (or agent), no matter how distant they are in the model hierarchy. The syntax of expressions allowing you to move up and down the model hierarchy and penetrate inside agents is explained in Section 10.9. Using those rules an agent can:

- Directly call functions of other agents
- Read and modify variables and parameters of other agents

In addition, AnyLogic supports a communication mechanism specific to agent based modeling: *message passing*. An agent can send a message to an individual agent or a group of agents. A message can be an object of any type and complexity, for example, a text string, an integer, a reference to an object, or a structure with multiple fields.

Synchronous and asynchronous communication

The fundamental difference between message passing and inter-agent function calls is that the first one is *asynchronous* communication, whereas the second one is *synchronous*. Consider Figure 3.44. The agent a send the message "Message" to the agent b by calling the function send() somewhere in the middle of event 1. The message is delivered to b, but execution of the reaction to the message is postponed until event 1 finishes and is performed in a new event 2, scheduled immediately after event 1.

Compare this to the function call. When a calls the function() of b, the execution of function() starts immediately in the middle of the event 1, the execution of agent a code is suspended and resumes only when function() returns control there. The same scheme applies when you use special functions deliver() and receive() as shown at the bottom of Figure 3.44.

Unless you intentionally want to use synchronous communication, we recommend using asynchronous message passing because it leads to a "cleaner" ordering of events, which is better for understanding and easier for debugging. Direct function

calls can cause complex chains and loops (for example, imagine what would happen if, in the middle of execution of the function(), the agent b calls another function of agent a), and this is not the kind of complexity a simulation modeler should spend time on.

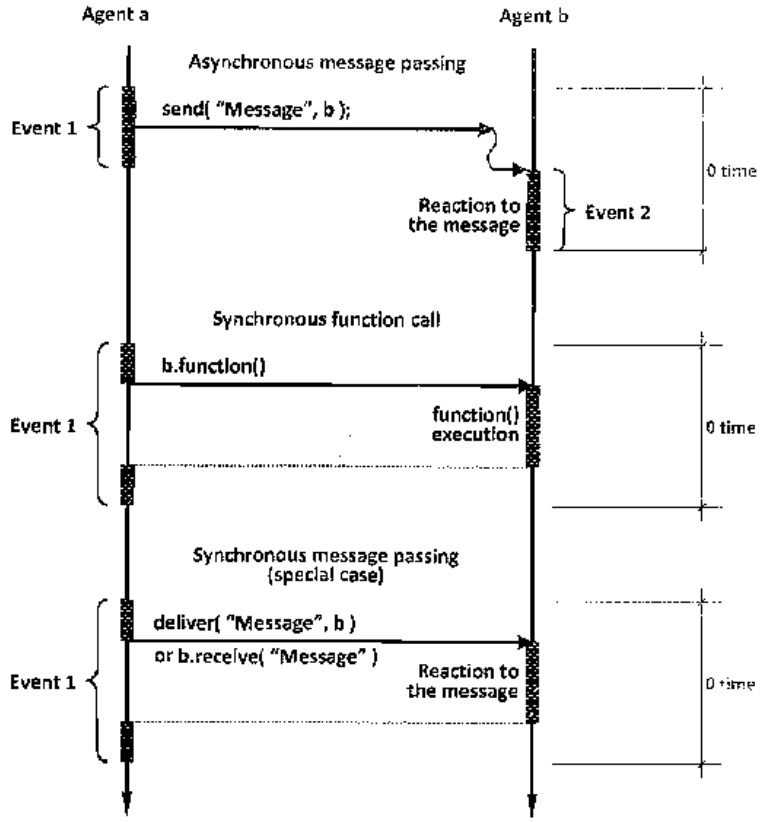


Figure 3.44 Message passing and function calls between agents

API for message passing

AnyLogic offers the following API for sending messages. These functions are available in the Agent class:

- **send(Object msg, Agent dest)** – sends the message to a given agent.
- **send(Object msg, int mode)** – sends the message to a single or a group of recipients, subject to the **mode** parameter.

- **deliver(Object msg, Agent dest)** – immediately delivers the message to a given agent and executes the recipient's reaction.
- **deliver(Object msg, int mode)** – immediately delivers the message to a single or a group of recipients (subject to **mode**) and executes their reactions.
- **receive(Object msg)** – immediately delivers the message to this agent and executes its reaction (this function is typically called by other agents).

The **mode** parameter is explained in Figure 3.45. **ALL** means all agents in the environment (not just agents of same type). **ALL_CONNECTED** delivers message to all agents connected to this one via standard connections. **RANDOM_...** chooses a random agent from a given set; the sender itself may be chosen.

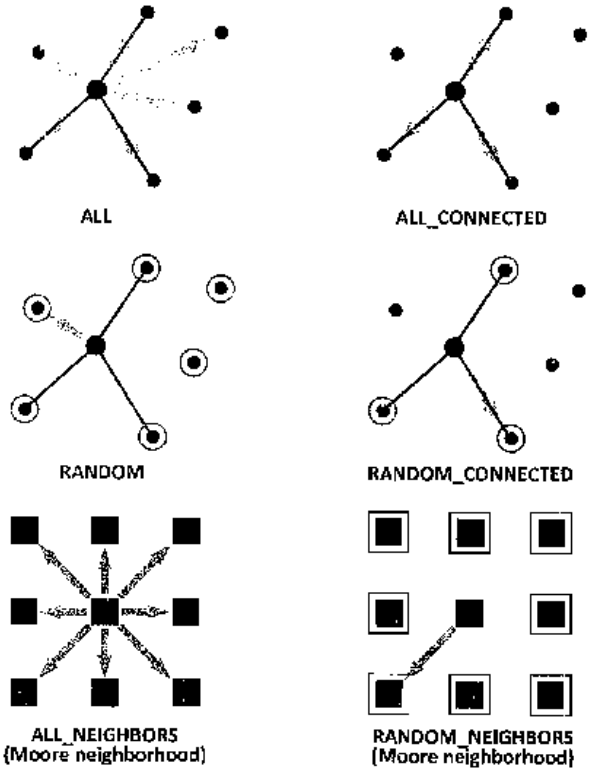


Figure 3.45 Different modes of message passing

You can also ask the environment to do the message passing by calling these functions:

- **deliverToAll(Object msg)** – immediately delivers a message to all agents in the environment.
- **deliverToRandom(Object msg)** – immediately delivers a message to a randomly chosen agent.

These functions are typically used when a message is sent not from one agent to another, but from some construct at the container level (e.g., from an event at Main) to an agent.

Message handling

An agent has a special code field **On message received** where you can define its reaction to the incoming messages, see Figure 3.46. In addition, all messages received by the agent are optionally forwarded to the statecharts inside the agent and can trigger transitions. Statecharts allow you to visually specify how the agent's reaction on a particular message depends on its internal state, and they are used more frequently than the **On message received** code.

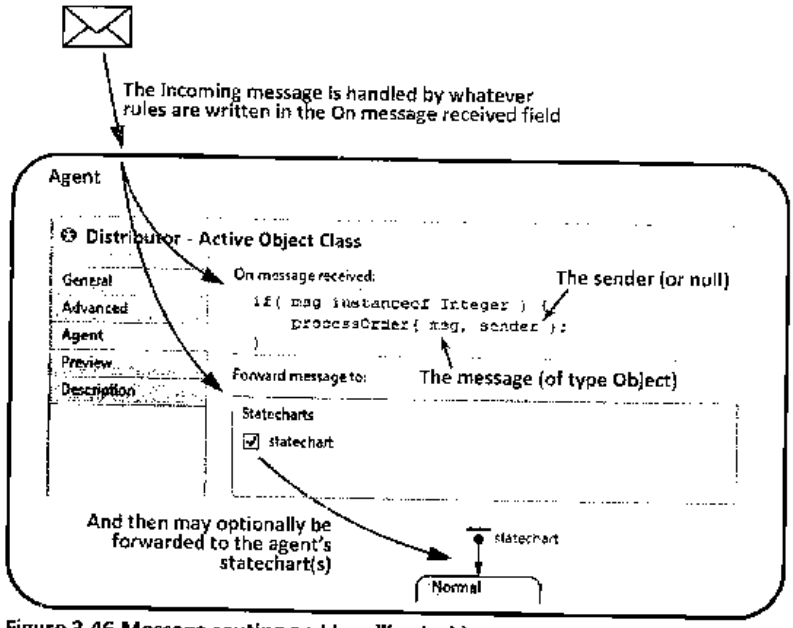


Figure 3.46 Message routing and handling inside an agent

Other types of inter-agent communication

As you know, agents can access variables and parameters of other agents. We, however, strongly encourage you limit this access to read-only type to keep

interface between agents "clean and thin". For example, it is absolutely fine to check if the other agent has certain properties:

```
if( otheragent.male and otheragent.income > 100000 ) ...
```

But be careful with modifying variables of other agents. If you choose to do so, and write something like this:

```
otheragent.income += 20000;
```

the income of **otheragent** will indeed grow, but it will not notice it immediately (and react) unless you explicitly call **otheragent.onChange();**.

3.9. Dynamic creation and destruction of agents

Agents and, in general, active objects in AnyLogic can be dynamically created and destroyed. The only object that has to be present in the beginning of the simulation, and cannot be deleted at runtime, is the "root" object, like **Main**. Creation and deletion of active objects is done using a *replicated object* construct – a collection of active objects of the same type. At runtime you can add more objects to the replicated object or delete existing objects.

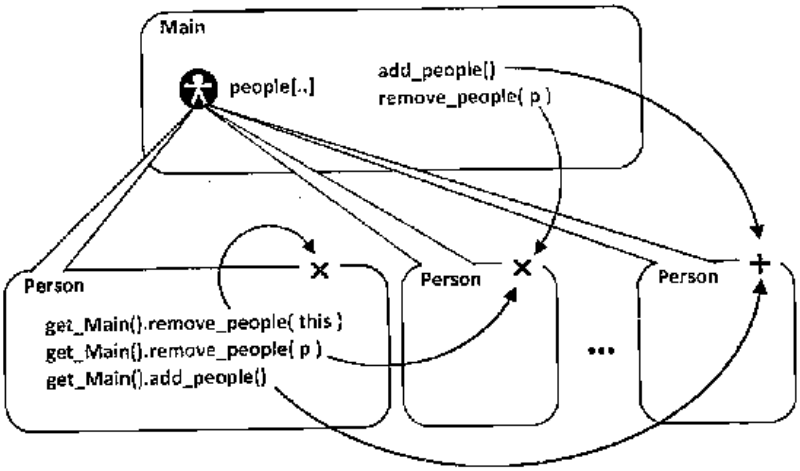


Figure 3.47 Using replicated object construct to dynamically create and destroy agents

If you need to create only one instance of an active object dynamically, you still need to use a replicated object with the initial number of objects set to 0.

▶ **If you plan to add or delete objects type Person within the object Main, you need to:**

1. Drag the icon of **Person** from the **Projects** tree to the editor of **Main**. This embeds an instance of **Person** into **Main**.
2. Select the **Replicated** checkbox in the properties of embedded object. Give the embedded replicated object a meaningful name, like **people**.
3. Specify the initial number of objects, possibly 0.

After you do it, AnyLogic prepares several useful functions for you. They are:

- The function **add_people()** of **Main**. This function creates a new object of type **Person**, places it into the collection **people**, and returns the newly created active object.
- The function **remove_people(Person object)**, also in **Main**. This function deletes a given instance in the collection people.
- The function **get_Main()** in the class **Person** returning the container of this instance of Person if it is embedded into **Main**. This function is actually generated for all classes whose instances are embedded in **Main**, be they replicated or not.

The replicated object provides API common for any Java collection, in particular:

- **int people.size()** – returns the number of active objects (or agents) in the collection.
- **Person people.get(int index)** – returns the element of the collection (the active object or agent) with the given index.
- **Person people.random()** – returns a randomly selected element.

▶ **To delete itself from the model, a Person object needs to:**

1. Call **get_Main().remove_people(this);**. The meaning of this code is: go one level up to access the container object of type **Main**, then call the function **remove_people()** available in **Main** and provide itself ("this") as a parameter.

▶ **Correspondingly, to create another person a Person object needs to:**

1. Call **get_Main().add_people()** or **Person newborn = get_Main().add_people()** if you need to remember the newly created person to do, for example, its additional setup.

There is a way to specify the parameters of the active object directly at the time of its creation. Assume the **Person** has two parameters: **male** of type Boolean and **income** of type double. Then an additional form of the function **add_people()** is generated:

```
add_people( boolean male, double income )
```

This creates a new object with the given parameters.

3.10. Statistics on agent populations

The easiest way of collecting statistics in agent based models in AnyLogic is to use the standard *statistics of the replicated object*. You can count the number of agents satisfying a certain condition (for example, being in a particular state), calculate the average of a certain numeric property across the agent population (for example, average income), and calculate minimum, maximum, and total (sum).

You can also create dynamic histograms displaying the *distribution* of a value throughout the population, such as age or income distribution.

In addition, as long as every internal element of every agent is accessible at the global (Main) level, you can collect highly customized statistics of arbitrary complexity. In this section we give some examples.

Example 3.9: Kinship model with standard statistics

We will take Example 3.8: "Kinship modeled using custom links" introduced earlier in this chapter and add several statistics, namely:

- Number of juniors, male adults, female adults, and seniors, and
- Average number of kids in a family

We will display the statistics using time stack charts and time plots.

▶ **Create statistics items in the replicated object:**

1. Open the model "Kinship modeled using custom links" described earlier. Open the editor of **Main** and select the replicated object **people** (the population of agents).
2. Open the **Statistics** page of the **people** properties and click **Add statistics.** A new statistics item is created.
3. Name the new statistics **nJunior** (n will mean "number of"). Leave the default type of the statistics (**Count**) and specify the condition:
   item.statechart.isStateActive(item.Junior)
   When this statistics is calculated, AnyLogic will iterate across the agent population, evaluate the condition for each agent, and return the number of times the condition evaluated to true. The **item** represents the current agent (person).

Note that the name Junior was defined within the **Person** class, therefore we need to provide the prefix "**item.**" (or "**Person.**") because we are now writing a piece of code "on the territory" of the **Main** object.

4. Create three more statistics items:
   nFemaleAdults that counts people with condition
   item.statechart.isStateActive(item.Adult) && ! item.male
   (this is a condition on both the current statechart state and a parameter),
   similar item **nMaleAdults** with condition
   item.statechart.isStateActive(item.Adult) && item.male
   and **nSenior** with condition
   item.statechart.isStateActive(item.Senior)
5. Finally, create the fifth statistics item with name **aveKids** (average number of kids). Change the type of this statistic to **Average** and specify:
   Expression: item.kids.size()
   Condition: ! item.statechart.isStateActive(item.Junior) && ! item.male
   This statistics will calculate the average value of the expression across those agents who satisfy the condition, namely for adult and senior females only.

We have created five statistics items, and AnyLogic has generated five functions in the replicated object people. Now we can write (while in **Main**):

- **people.nJunior()** to obtain the current number of juniors (integer value).
- **people.nFemaleAdults()** to obtain the current number of female adults.
- **people.nMaleAdults()** to obtain the current number of male adults.
- **people.nSenior()** to obtain the current number of senior people.
- **people.aveKids()** to obtain the average number of kids per non-junior female (a real value).

So far, nobody is calling these functions and statistics are not collected. We will now add a couple of charts and let them periodically update themselves by calling the statistics.

▶ **Add charts displaying the statistics:**

6. While still in the editor of **Main**, add the **Time stack chart** object from the **Analysis** palette.
7. In the properties of the chart, add a new data item with default type **Value**, title **Junior**, and value **people.nJunior()**. Set the color of this item to **yellowGreen** to match the color of junior agents.
8. Similarly, add three other items: **Female adults**, **Male adults**, and **Senior**.

Look at the bottom part of the chart properties. The chart is set to update itself each 1 time unit (each year in our model), to keep 100 last data items, and to display the 100 years time window. This means the statistics functions will be called by the chart each year.

9. Add a **Bar chart** with the data item **Ave no of kids** and value **people.aveKids()**. This chart will also update itself each model year.
10. Run the model and watch the charts.

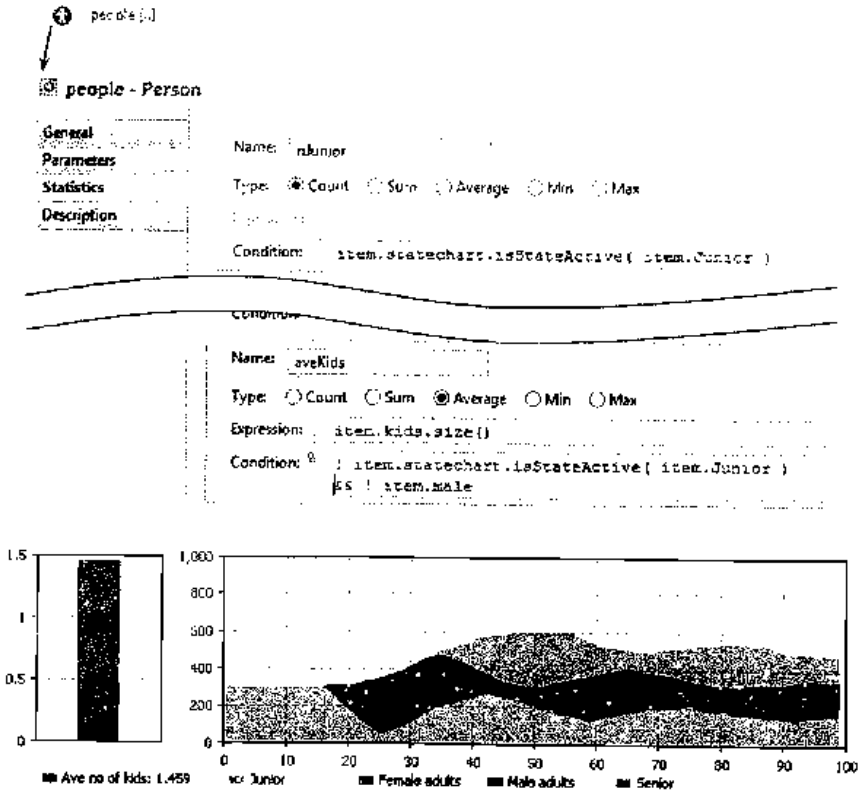


Figure 3.48 Standard statistics in the Kinship model

Example 3.10: Kinship model with dynamic histograms

We will now further develop the statistics collection in the Kinship model. The statistics we have collected and displayed so far are useful, but we want to know more things about the population, for example, the age distribution or the distribution of the number of kids per family. The best way of answering such questions is to maintain and display *dynamic histograms* reflecting the current distribution of a given value.

To be able to display the age distribution, we will need to slightly modify the model of **Person**. In the initial implementation, the exact age of a person is actually lost: we only know the current age group (junior, adult, senior), and the time remaining before the person leaves the group (the remaining time of a **timeout** transition). We will now add a variable **birthdate** and the function **age()**.

▶ **Add explicit age information to Person:**

1. Open the model "Kinship model with standard statistics", and open the editor of **Person**. Drag a **Variable** object from the **General** palette, name the variable **birthdate**, leave the default type (**double**), and set the initial value to **time()**. Now, when a new person is created, the **birthdate** is set to the current model time.
2. Add a function **age()**, returning type **double**, with the code return **time() - birthdate;**. This function will return the current age of the person.

▶ **Create periodically updated distributions**

3. Open the editor of **Main** and add a **Histogram data** object from the **Analysis** palette. Name the item **ageDistribution**, set the **Number of intervals** to 20, set the **Values range** to 0-75, and select the option **Do not update data automatically**.
4. Create a copy of the histogram with the name **kidsDistribution** and the same properties except for the **Values range**, which should be 0-10.
5. Create an event **updateDistributions** (the **Event** object can be found in the **General** palette) and set its mode to **Cyclic Timeout** with **Recurrence time** 1. In the **Action** of the event write:

```
//clear previous data
ageDistribution.reset();
kidsDistribution.reset();
//update histograms
for( Person p : people ) { //for all agents in the population
  ageDistribution.add( p.age() ); //add person's age to age distribution
  if( !p.statechart.isStateActive( p.Junior ) && !p.male ) //female non-junior only
    kidsDistribution.add( p.kids.size() ); //add no of kids to kids distribution
}
```

▶ **Display the distributions using Histogram objects:**

6. Add a **Histogram** chart from the **Analysis** palette, click **Add histogram data** in its properties, specify title "Age distribution", and write **ageDistribution** in the **Histogram** field.
7. Add another histogram chart with the title "No of kids distribution" displaying the **kidsDistribution** correspondingly.
8. Run the model. Now the statistics are much more informative, see Figure 3.49.

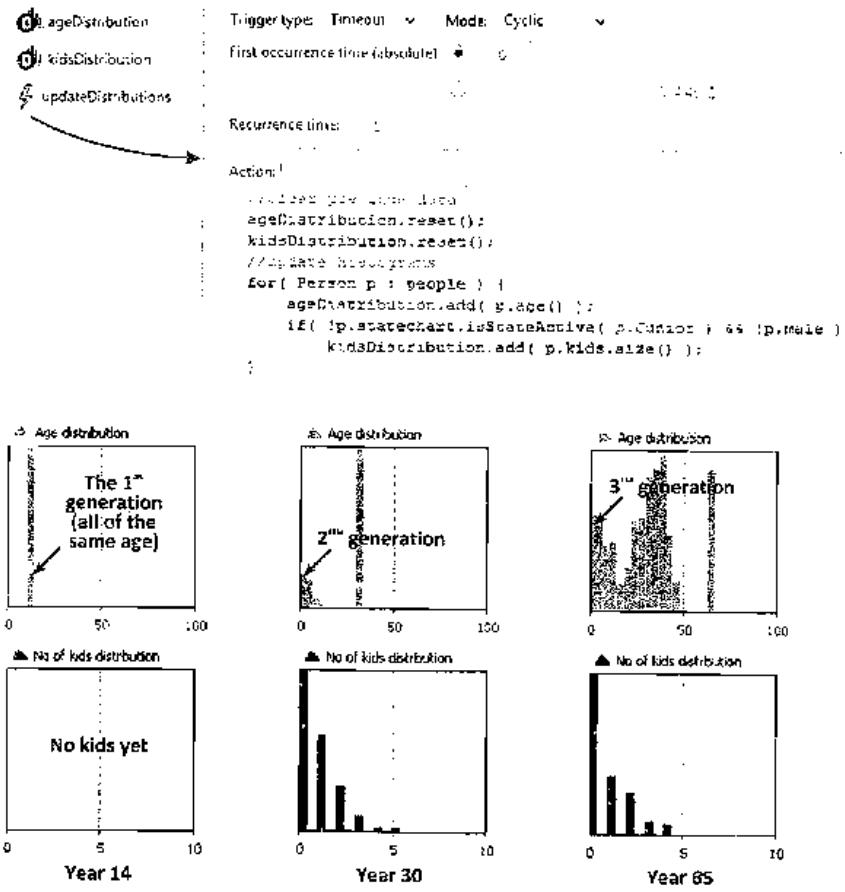


Figure 3.49 Dynamic histograms in the Kinship model

Customized high performance statistics

As you can see, the standard statistics collection involves iteration across the entire population of agents each time the value is updated. This may be computationally inefficient, especially when the population is large. If the simulation performance becomes an issue, you may consider replacing the standard statistics with customized high performance statistics. The general idea is to use the knowledge of the model to update the statistics only when relevant changes occur.

For example, if you are interested in the number of infected people in an epidemic model, you can maintain an integer counted at the top (Main) level and let the agents-patients increment it each time a patient gets infected and decrement when

the patient recovers. In a population model, where you are interested in average income per capita, you can add a variable holding the total income of the entire population and change it each time an individual person's income grows or falls. Then, to obtain the average income, you only need to divide the total income by the population size.

Example 3.11: Kinship model with customized statistics

We will refactor the model "Kinship model with standard statistics" by replacing the statistical items of the replicated object with plain variables directly updated by the agents.

▶ **Remove all standard statistics items and create auxiliary variables:**

1. Open the previously created model, select the **people** object in the editor of **Main**, and delete all items on the **Statistics** page.
2. In Main, add four variables of type **int**: **nJunior, nFemaleAdults, nMaleAdults, nSenior.** These will be the counters.
3. Add two more variables of type **int**: **totalKids** and **nFemaleNonJunior.** These two variables will help us to obtain the average number of kids per family.
4. In the time stack chart, change the expression **people.nJunior()** to simply **nJunior**, **people.nFemaleAdults()** to **nFemaleAdults**, etc.
5. In the bar chart, change the expression **people.aveKids()** to **(double)totalKids / nFemaleNonJunior**. (We have to explicitly convert one of the operands to the type **double** to avoid integer division.)

▶ **Program updates of the statistical variables:**

6. Open the editor of **Person**. Select the statechart state **Junior** and add this code line to the **Entry action**: **get_Main().nJunior++;** and write **get_Main().nJunior--;** in the **Exit action**.
7. Similarly, in the **Entry action** of the state **Adult** add:

```
if( male ) {
  get_Main().nMaleAdults++;
} else {
  get_Main().nFemaleAdults++;
  get_Main().nFemaleNonJunior++;
}
```

   and in the Exit action, correspondingly:

```
if( male )
  get_Main().nMaleAdults--;
else
  get_Main().nFemaleAdults--;
```

8. In the **Entry action** of the state **Senior**, add this line:
   get_Main.nSenior++.

and in the **Exit** action type:

```
get_Main().nSenior--;
if( !male )
  get_Main().nFemaleNonJunior--;
```

9. Select the transition **NewKid** and add this line to its **Entry** action:

```
get_Main().totalKids++;
```

10. And finally, modify the **Action** of the **Dead** state this way:

```
//clean up all links to me
//from kids
for( Person kid : kids ) {
  if( male ) {
    kid.father = null;
  } else {
    kid.mother = null;
    //update stats (mother dies first)
    get_Main().totalKids--;
  }
}
//from spouse
if( spouse != null)
  spouse.spouse = null;
//from parents
if( father != null )
  father.kids.remove( this );
if( mother != null ) {
  mother.kids.remove( this );
  //update stats (child dies first)
  get_Main().totalKids--;
}
//and die
get_Main().remove_people( this );
```

11. Run the model. Switch to the Virtual time mode and watch how fast the simulation is after the changes that we made.

As you can see, the simulation performance comes at the cost of writing some code, which may not be trivial. In this particular example, the most challenging part is to maintain an up-to-date total number of kids. As long as we originally decided to count children of adult and senior females, we have to take care of decrementing the total number of kids when either the mother dies before her kids, or a child dies before his or her mother. Otherwise, the code is pretty straightforward.

3.11. Condition-triggered events and transitions in agents

AnyLogic allows you to specify events and transitions triggered by a condition – Boolean expressions defined over the agent's local and possible external variables.

Such events or transitions should occur once the associated condition becomes true. It makes sense to discuss the AnyLogic implementation of these constructs.

We distinguish between the two cases:

- **The model contains dynamic variables and equations** (for example, the model includes system dynamics components). Then the numeric solver is involved and the implicit (and typically small) *time steps* are present in the model. If this is the case, all conditions of events and transitions *are tested on every numeric time step*, and the moment when a condition becomes true will be discovered *at least with the time step accuracy*.
- **The model does not contain dynamic variables and equations**. Then the numeric solver is not working during the simulation, and there are no implicit time steps in the model. The conditions the agent's events and transitions wait on are then *tested only when something happens in the agent* (e.g., an internal event or statechart state change), and also *when the agent's function* **onChange()** *is explicitly called*.

Therefore, it makes a lot of sense to explicitly notify the agent that is waiting on a condition each time an external change occurs that may affect the condition. For example, if the agent-consumer waits for the product price to fall below a certain threshold, the agent-retailer may call the function **onChange()** of all consumers in the beginning of a sale. The model constructed this way will be computationally very efficient.

However, the model structure and behavior does not always allow it to be done in a simple way, and then the modeler can introduce "artificial" time steps by, for example, adding a cyclic event at the **Main** object and calling **onChange()** of all agents in the action of that event.

Chapter 4. How to build agent based models. Field service example

Agent based modeling is the easiest modeling method. You identify which objects in the real system are important for solving the problem, and create those same objects in the model. You think of the behavior of those objects relevant to the problem, and program that same behavior in the model. You do not have to exercise your abstraction skills as much as in system dynamics or force yourself to "always think in terms of a process," as in discrete event modeling.

Described in one page, the process of building an agent based model includes answering the following questions:

1. Which objects in the real system are important? These will be the agents.
2. Are there any persistent (or partially persistent) relationships between the real objects? Establish the corresponding links between the agents.
3. Is space important? If yes, choose the space model (2D, 3D, discrete) and place the agents in the space. If the agents are mobile, set velocities, paths, etc.
4. Identify the important events in the agents' life. These events may be triggered from outside, or they may be internal events caused by the agent's own dynamics.
5. Define the agents' behavior:
   5.1. Does the agent just react to the external events? Use message handling (see Section 3.8) and function calls (see Section 10.4).
   5.2. Does the agent have a notion of state? Use a statechart (see Chapter 7).
   5.3. Does the agent have internal timing? Use events (see Section 8.2) or timeout transitions (see Section 7.3).
   5.4. Is there any process inside the agent? Draw a process flowchart.
   5.5. Are there any continuous-time dynamics? Create a stock and flow diagram (see Section 5.1) inside the agent.
6. Do agents communicate? Use message sequence diagrams (see Section 4.2) to design communication/timing patterns.
7. What information does the agent keep? This will be the memory, or state information, of the agent. Use variables (see Section 10.3) and statechart states.
8. Is there any information, and/or dynamics, external to all agents and shared by all agents? If yes, there will be a global part of the model (the term "environment" is sometimes used instead).

9. What output are you looking for? Define the statistics collection at both the individual and aggregate levels.

In this chapter, we give a very detailed description of a model design process, from the very beginning to the optimization results and assumptions discussion.

4.1. The problem statement

A fleet of equipment units (for example, wind turbines or vending machines) are distributed geographically within a certain area. Each equipment unit generates revenue while it is working. However, it sometimes breaks down and needs to be repaired or replaced. Maintenance is due every maintenance period. Late maintenance, as well as advanced age, increase the probability of failure.

Figure 4.1 Wind turbines

The service system consists of a number or service crews that are based in a single central, or "home," location. When a service or maintenance request is received by the service system, one of the crews takes it, drives to the equipment in question, and performs the required work. During the failure examination, it may turn out that the equipment cannot be repaired, in which case it is replaced. If already due, any currently scheduled maintenance is done after the repair, within the same visit. (The service crew can also replace aged equipment even if it is still working, subject to the replacement policy.) Having finished the work, the service crew may take

another request and drive to the next unit location, or, if there are no requests, return home.

A service crew has constant daily costs associated with it. Each operation (maintenance, repair, or replacement) has an additional one-time cost. All parameters of the system are listed in the Table.

Parameter	Value
Daily revenue per working equipment unit	$400
Daily cost of a service crew, including driving costs	$1,500
Average repair cost	$1,000
Maintenance cost	$600
Replacement cost	$10,000
Typical repair time	5 hours
Typical maintenance time	3 hours
Typical replacement time	12 hours
Probability replacement needed after failure	10%
Maintenance period	90 days
Base failure rate (maintenance done, equipment not aged)	3 / 100 days
Service crew mobility	500 miles / day
Equipment units in the fleet	100
Area serviced	600 x 500 miles

The goal of this model is to find the number of service crews, and the replacement policy, that result in maximum profit for the equipment fleet.

4.2. Phase 1. Can be done on paper

Who are the agents?

As we said earlier, mapping the problem to the modeling language is easy in the agent based modeling paradigm. What objects in the real system do we observe and are interested in? Equipment and service crews. So, these will be the agents in our model. Do we need a central service dispatcher of any kind, or any environment objects? We will find out later during the model building process.

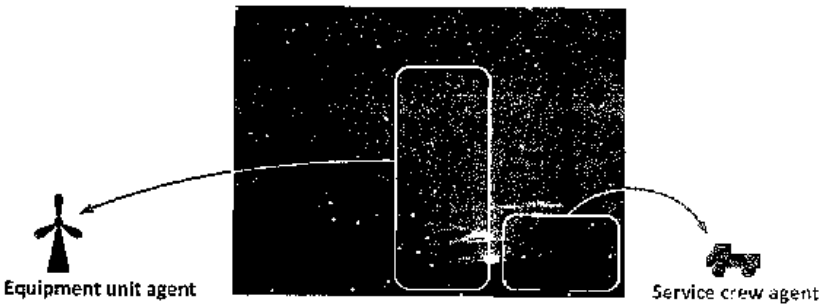


Figure 4.2 Agents in the field service model

Equipment unit agent

We definitely do want to know if the equipment is working or not. So we will distinguish between at least two *states* of the equipment unit – Working and Not working, – and will try to use the *state transition diagram* to model the equipment behavior, see Figure 4.3 version 1.

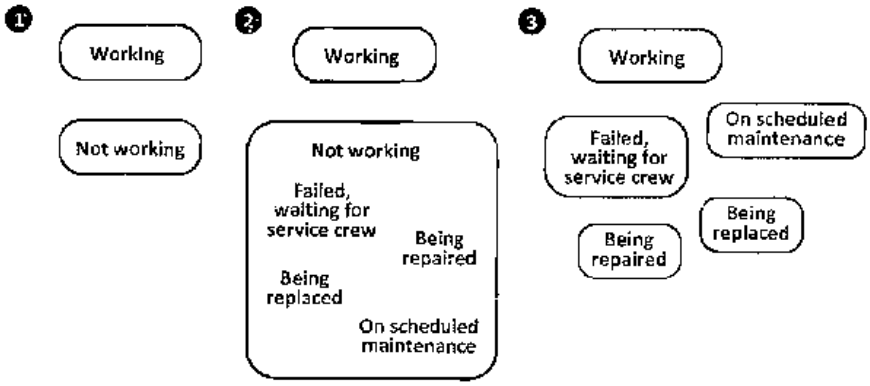


Figure 4.3 State transition model of the equipment unit, versions 1, 2, and 3

While the equipment is not working, it can be in failure (and be waiting for the service crew to arrive and fix it), be repaired or replaced, or be on scheduled maintenance, Figure 4.3 version 2. Are these separate states or one state? Making these three separate states makes sense if:

- The behavior of the equipment unit is essentially different in different states – say, different time spent in the state, different reactions to the external events, or different actions taken upon exiting the state; or
- We want to have separate statistics for the states.

In our case, all arguments are for separate states. We do want to collect state-based statistics (for example, find out the average waiting time for the service crew); time of operations is different; and behavior is different (for example, maintenance can be done directly after repair, but not after replacement). So, we end up with five states, see Figure 4.3 version 3.

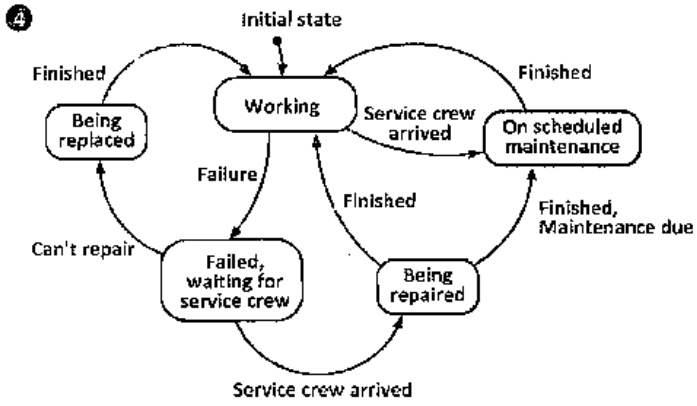


Figure 4.4 State transition model of the equipment unit, version 4

Next, let us think of how the equipment *changes* its state. Initially, we will assume the equipment is **Working**. In the event of failure, the equipment transitions from the **Working** state to the **Failed** state. Then, after the service crew has arrived and examined the equipment, the state changes either to **Being replaced** or **Being repaired**. When this is complete, the equipment returns to the **Working** state. However, there is one exception: after repair, it may so happen that maintenance is due; in that case, maintenance is done immediately. Also, the service crew can go to the working equipment for scheduled maintenance. Now we have a tentative *state transition diagram*, see Figure 4.4.

To complete the diagram, we need to determine *how the transitions are triggered*. **Failure** is clearly a stochastic event internal to the equipment. Both replacement and repair start upon the arrival of a service crew, so in fact, there is *only one* transition from the **Failed** state triggered by the crew arrival. This transition has two branches. One leads to **Being repaired**, and another to **Being replaced**. The decision is made internally, and is probabilistic according to the problem statement. Similarly, the transition from the **Being repaired** state taken upon repair completion has two branches as well. This time, the decision is deterministic: if maintenance is due, the next state is **On scheduled maintenance**; otherwise, the next state is **Working**. The

transitions labeled **Finished** are all time-driven, the time spent in a state corresponding to work being done can be obtained from the input data.

Notice that, depending on the state, the equipment unit reacts differently to the arrival of the service crew. If the crew comes to the working equipment, it can only mean it will do the scheduled maintenance, whereas, in the **Failed** state, it will examine the situation and repair or replace the equipment. It is also worth mentioning that, according to the semantics of statecharts (see Chapter 7), the transitions are *instantaneous* (take zero time), so in our statechart we assume the examination of failure takes zero time. This is a fair assumption: if we want, we can simply add this time to the duration of both replacement and repair.

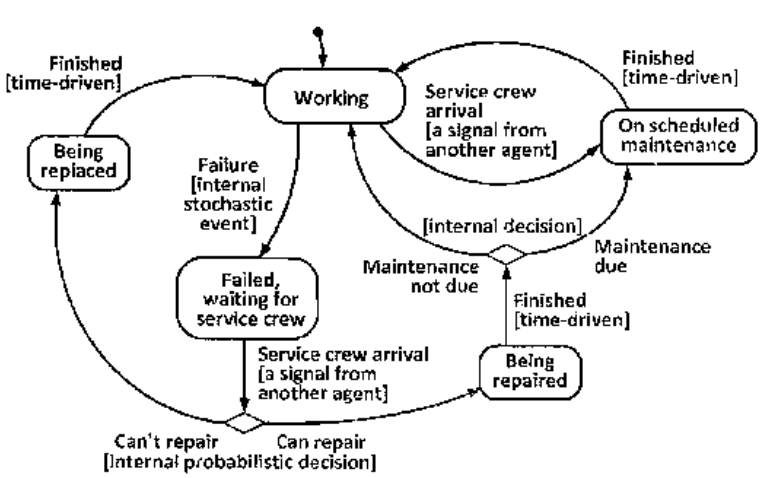


Figure 4.5 State transition model of the equipment unit, version 5

That's it for the equipment unit so far. We will add more details after we make some progress with the other parts of the model's design.

Service crew agent

The service crew behavior is simple: it takes a request, drives to the equipment unit, does the work, and either continues with the next request or returns to its home location. Unlike the equipment unit, the service crew is a mobile agent, so moving in space will be the essential part of its behavior. The notion of state, however, (**Idle**, **Driving**, **Working**) seems to be present here as well, so we will follow the same design pattern we used for the equipment unit, see Figure 4.6 version 1.

If we draw all possible states transitions, we will get what is shown in Figure 4.6 version 2. There are two transitions from the **Driving** state, both triggered by **Arrival**.

One leads to **Idle**, and the other to **Working**. The destination state depends on where the crew was driving. How do we capture this in the statechart? One way is to check the location upon arrival. Another, more elegant method is to have two different states for driving: **Driving to work** and **Driving home**. The decision of where to drive is made when the current task is completed (transition **Finished** triggered by the equipment's timing) and depends on whether or not there are more requests waiting to be serviced. This can be modeled by a construct we are already familiar with: a transition with two branches, see Figure 4.6 version 3.

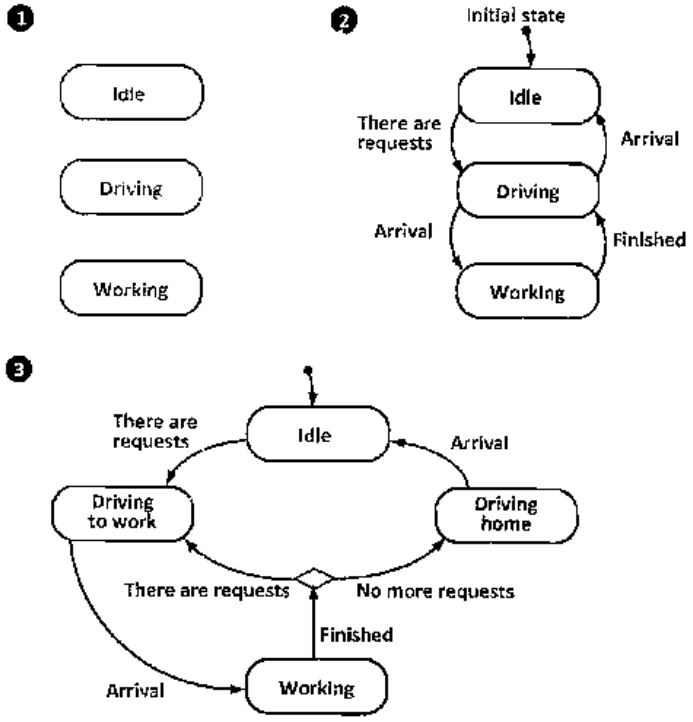


Figure 4.6 State transition model of the service crew, versions 1, 2, and 3

In the resulting statechart there are two arcs labeled **There are requests**: one is from **Idle** state, and the other is a branch of the **Finished** transition. While in the latter case the condition is *tested once* upon completion of the current work assignment, in the former case the condition should be *constantly monitored* while the service crew is idle. We will combine them into one: in **Idle** the service crew will be *explicitly told* to check if there are requests, see Figure 4.7 version 4. The reason for making this change in this case technical, based on the knowledge of agent based modeling.

Multiple service crew agents may check the request queue at the same time, but there may be fewer requests than agents, so some will return to the Idle state (in our statechart: through the Driving home state, which will be passed through immediately in case the agent is already home).

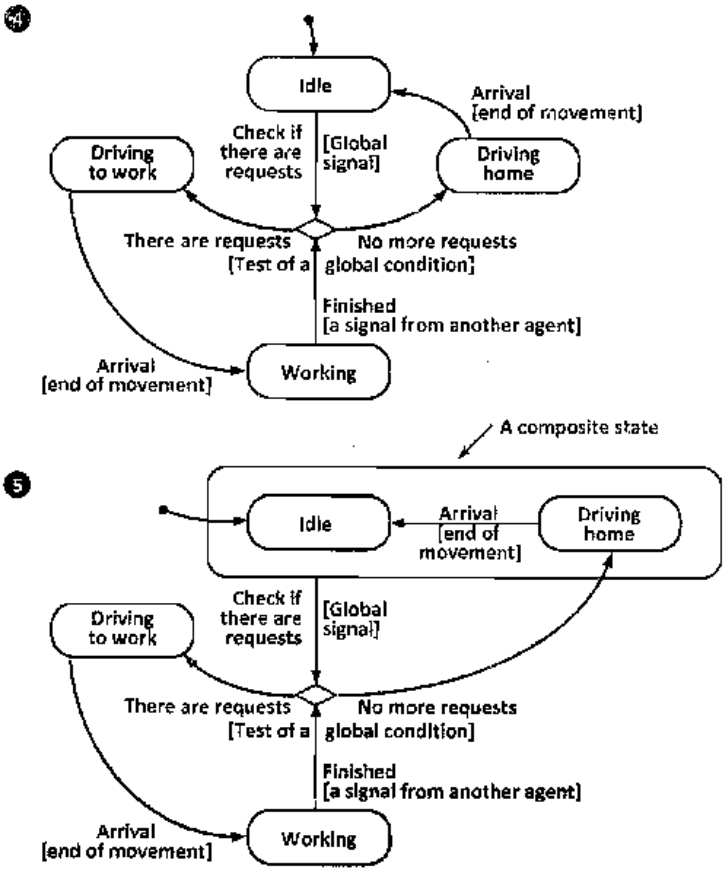


Figure 4.7 State transition model of the service crew, versions 4 and 5

However, it may so happen that the notification about new requests comes while the service crew is driving home. In that case the crew will ignore it and, having arrived at the home location, will stay there, idle (until, most likely, another piece of equipment breaks down). This is no good. We can assume the service crew is equipped with a radio and can take requests while driving. So, if the "check request" signal comes in the Driving home state, the crew will accept it, change the route, and drive to the equipment unit. In the statechart we could draw another transition

triggered by the signal from the **Driving home** state to the decision diamond, but we will use a great statechart feature – a composite state (see Section 7.2).

A composite state is a group of states that have some common behavior – for example, the same reactions to events, or common timeouts.

The updated statechart of the service crew agent is shown in Figure 4.7 version 5. The transition **Check if there are requests** exits the composite state, and therefore applies *both* to the **Idle** and **Driving home** states. Note that transitions may freely cross the composite state borders; for example, the branch **No more requests** directly enters the **Driving home** state, and the initial state pointer points to the **Idle** state.

While designing the service crew behavior, we were constantly referencing the "request queue" and assuming somebody can "tell all agents" to check it out. The request queue is clearly global for all agents, as well as the central "dispatcher" that takes care of it. We will return to those items later on.

Agent communication. Message sequence diagrams

During the design of the service crew behavior, we used reasoning that goes like, "What will happen if a notification about the new requests comes while the service crew is driving home?" We have already started thinking about the agent communication and event ordering. In this section, we will introduce a graphical notation that greatly helps in communication design. This notation comes from the computer science world; namely, from the distributed systems area.

A good agent based modeler should know a little bit about the *design of distributed and parallel systems*. An agent based model is, in fact, a pseudo-distributed system: it is a collection of objects that are concurrently active and communicate with each other (despite the concurrency is simulated by the engine). Deadlocks, livelocks, unexpected simultaneous event ordering, and other problems specific to distributed systems are possible in agent based models.

Let's list all communication instances that we have used in our models of the equipment units and the service crews. They are:

- **Service crew arrival**: a message from a service crew to an equipment unit.
- **Finished**: a message from an equipment unit to the service crew.
- **Check if there are requests**: a message from a "central dispatcher" to all service crews.

The best design notation for communication in concurrent systems is a *message sequence diagram*. In such a diagram, each object is represented by a vertical line, and messages (signals, function calls) from one object to another as horizontal

arrows. Time flows downward. Time can be both physical (continuous) and logical (representing just event ordering with possibly zero intervals between events). Individual events local to an object can also be placed on a message sequence diagram.

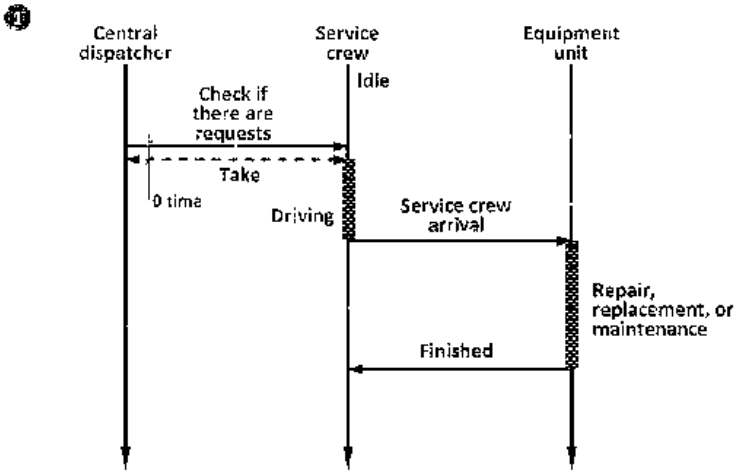


Figure 4.8 Message sequence diagram of the field service model, version 1

The first version of a message sequence diagram for our model is shown in Figure 4.8. There is just one equipment unit and one service crew. The shaded bars represent processes that take place inside the agents, and may end up with a message sent to another agent. Of course, both the service crew and the equipment unit are involved in the repair process; however, according to our model, it is the equipment unit that defines the duration or repair, and then notifies the service crew of completion.

However, in Figure 4.8 it is not clear who initiates the whole thing. We will now extend the diagram to include the equipment failure event. We will also add another equipment unit.

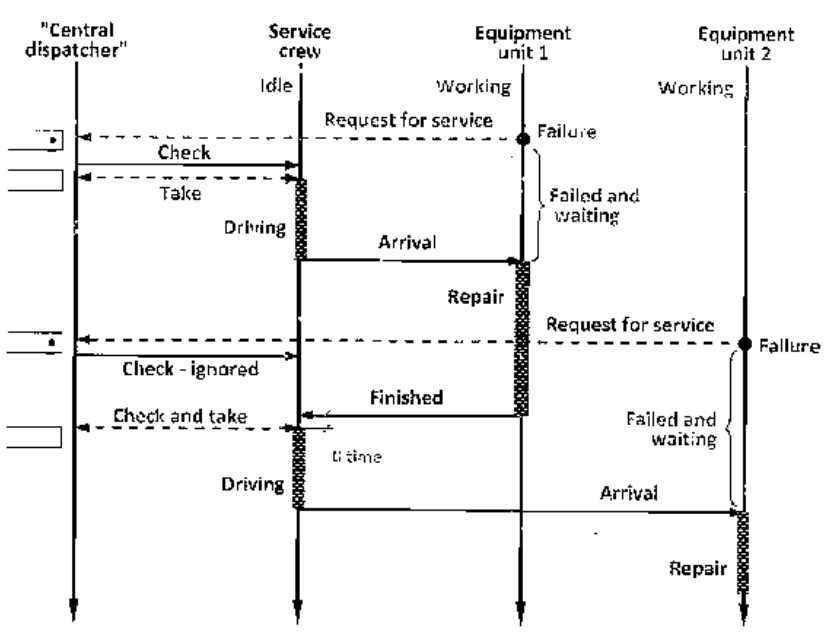


Figure 4.9 Message sequence diagram of the field service model, version 2

In version 2 of the message sequence diagram, we see the initial states of the agents: both equipment units are working, and the service crew is idle. Then, equipment 1 fails and posts a request to the "dispatcher." The "dispatcher" notifies the service crew, which immediately takes the assignment and starts driving to the failed equipment unit. While the first equipment unit is being repaired, failure occurs at the second unit. The request is posted and the "dispatcher" notifies the service crew, but this notification is ignored; the service crew is in the Working state and will not react to Check request queue messages, see Figure 4.9. The service crew will not test the request queue and take another assignment until the first repair is completed.

Message sequence diagrams, in our case, are an auxiliary design document intended to help the modeler with understanding agent communication. They are not formal or executable, and they are not a part of the AnyLogic modeling language.

The last communication scenario we will draw is one with one service request and two crews, see Figure 4.10. The "dispatcher" broadcasts the notification to all service crews. Both are idle, so both will check the queue. The events of checking

the request queue by the two agents are *simultaneous, but ordered*. Only one will take the assignment and start driving, while another will return to the Idle state.

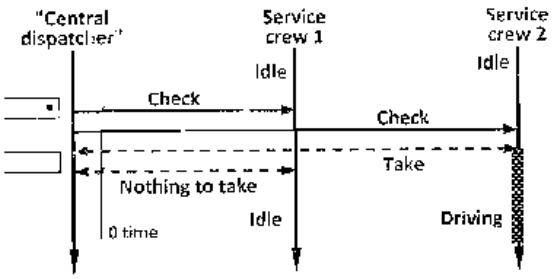


Figure 4.10 Message sequence diagram of the field service model. Two service crews

Space and other things shared by all agents

Among the things shared by all agents is the space where equipment is located and service crews drive. With that in mind, we will define a 2D space of 600 by 500 model miles, distribute the equipment evenly across that space, and place the service crews initially into a home location also within that area, see Figure 4.11. According to our simplified problem statement, we do not model roads; we just assume that, on average, a service crew can cover 500 miles in any direction if it drives for 24 hours. Thus, we will set up the speed for the service crew agents to 500 miles per day, and let them drive along straight lines directly from origin to destination.

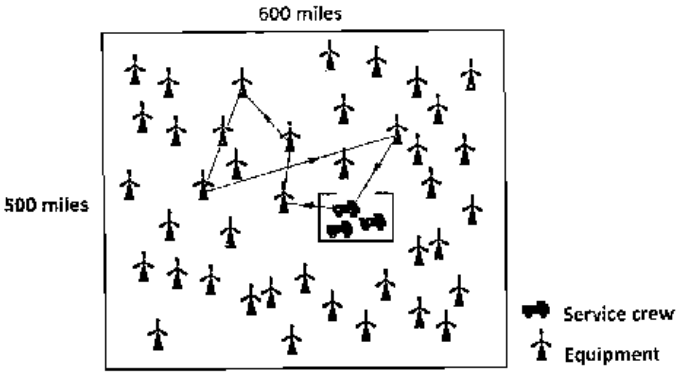


Figure 4.11 Space in the field service model

Apart from space at the global level, there are:

- The queue of service requests
- The "dispatcher" that manages the queue

How do we model this? Is the dispatcher active or passive? Does it have state or timing? The quick analysis of all we have said about it shows that the dispatcher does just one thing: it inserts a message in the queue and immediately broadcasts the notification to all service crews. *No state or timing.* Just stimulus and immediate response. This is best modeled as a *function call*. In addition, we can write functions that check the queue and remove the first request in the queue. See Figure 4.12 for this part of the model.

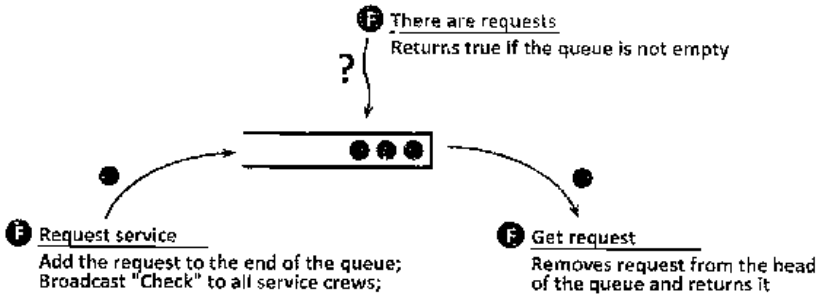

There are requests
Returns true if the queue is not empty
?
Request service
Add the request to the end of the queue;
Broadcast "Check" to all service crews;
Get request
Removes request from the head
of the queue and returns it

Figure 4.12 The queue of service requests and the "service dispatcher"

That's it for the first design phase. We have a sufficient level of detail to move the design to AnyLogic and run the model.

4.3. Phase 2. The model in AnyLogic. The first run

The model structure and the top level object Main

Mapping our design to the AnyLogic modeling language is straightforward. Equipment unit and service crew become agent classes **EquipmentUnit** and **ServiceCrew**, and all global things (space and request queue management) will be defined within **Main** – the top level object of the model. 100 equipment units (100 instances of **EquipmentUnit** class) and 3 service crews (just for example's sake) will be embedded into **Main**. Space and layout settings are defined in the **Environment** object, also in **Main**. Main will look like Figure 4.13.

The queue of requests is modeled by AnyLogic collection of type linked list (see Section 10.6) with elements of class **EquipmentUnit**.

Notice that we do not use a special data type for a service request – we just use the reference to the equipment unit that originates the request. This is good (because we follow the principle of minimalism), but it is only possible until we need to add more attributes to the request, such as timestamp, priority, or severity of problem.

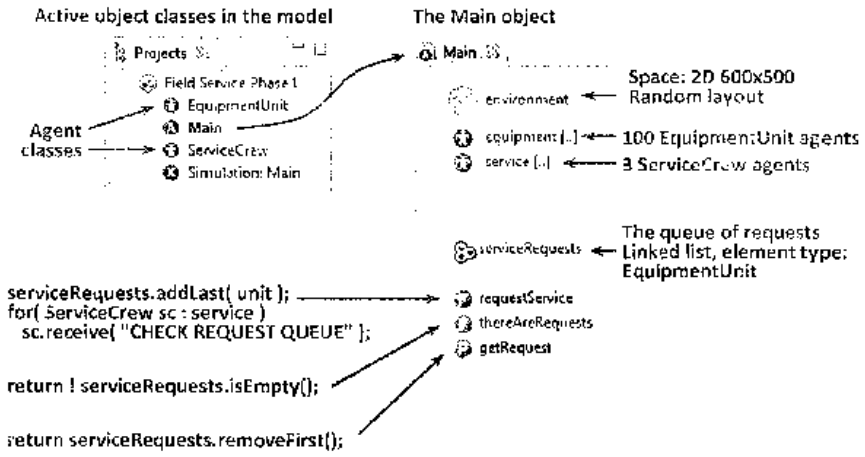


Figure 4.13 The structure and the top level of AnyLogic Field service model

The function **requestService(EquipmentUnit unit)** adds the equipment unit provided as the argument to the end of the queue and iterates through all the agents in the **service** to deliver to them the message **"CHECK REQUEST QUEUE"** (the agent's function **receive()** immediately delivers the message to the agent). The **boolean** function **thereAreRequests()** calls the collection's method **isEmpty()** and returns its negation. The function **getRequest()** removes the last element of the queue and returns it.

The EquipmentUnit agent

The **EquipmentUnit** active object class is declared as agent. On the **Agent** page of its properties, the space type is set to **Continuous 2D** and the checkbox **Environment defines initial location** is selected.

The AnyLogic statechart of the equipment unit agent is shown in Figure 4.14.

We encourage modelers to use colors to visually emphasize the meaning of states; for example, red for failure, green for up and running, etc. The same color scheme can then be used for statistics visualization.

Having drawn the statechart, we can specify the transition triggers (see Section 7.3). We have four transitions in this statechart that are time-driven (taken after the equipment unit has spent a certain time in a state):

- **Failure** – time to failure depends, according to our problem definition, on many factors. For now, we will just set the trigger type to **Rate**, and rate to the **BaseFailureRate**, which we will make a parameter. The transition of rate type

works in the same way as the transition with exponentially distributed timeout.

- **FinishRepair** – we know the typical repair time and will assume a triangular distribution. The timeout for this transition will be
  triangular(RepairTime * 0.5, RepairTime, RepairTime * 2.5)
- **FinishReplacement** – similarly, stochastic timeout:
  triangular(ReplacementTime * 0.5, ReplacementTime, ReplacementTime * 1.5)
- **FinishMaintenance** – same as above:
  triangular(MaintenanceTime * 0.5, MaintenanceTime, MaintenanceTime * 1.5)

There are two remaining transitions that are triggered by a message received by the agent from another agent: **SCArrivedForRepair** and **SCArrivedForMtce**. We could use a text message like "SERVICE CREW ARRIVED", but we need to *remember the service crew in the equipment unit* because, at the end of work, it will be the equipment unit who notifies the service crew that it can leave. Therefore, we will use the reference to the service crew as the message type and save it in a local variable **serviceCrew**.

To establish a temporary link between agents, you can either call the agent's function **connectTo()** or simply remember a reference to another agent in a variable. The advantage of a variable is that we can explicitly specify the type of another agent and can identify the kind of relation in case there are many (like best friend, parent, colleagues). Don't forget to delete the reference (set the variable to **null**) when the relationship ceases to exist.

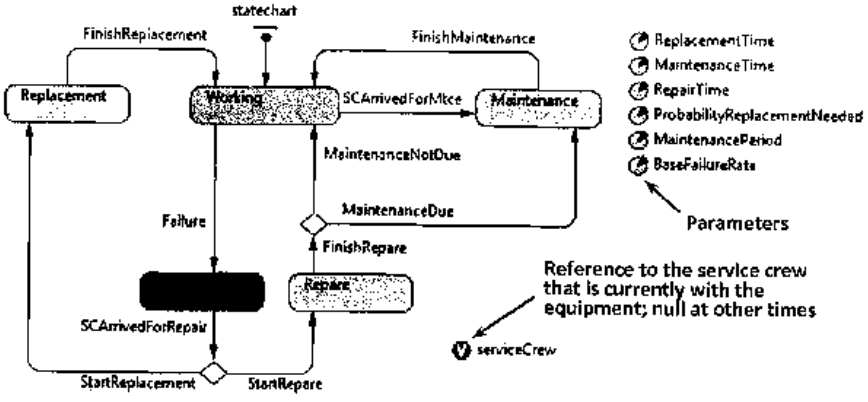


Figure 4.14 AnyLogic statechart of the EquipmentUnit agent

Two transitions in our model have branches; namely, **SCArrivedForRepair** and **FinishRepair**. Branch (see Section 7.3) can be considered as an "if" statement executed when the transition is taken. The condition of the branch **StartReplacement** is

probabilistic: we know that, with a certain probability, the failed equipment cannot be repaired and needs to be replaced. In the AnyLogic language, this can be written as `randomTrue( ProbabilityReplacementNeeded )`. The alternative branch **StartRepair** will be set as the *default* and will be taken if the condition of **StartReplacement** evaluates to false. At the moment, we cannot write the condition for the **MaintenanceDue** branch because we do not know the date of the last maintenance; moreover, we have not yet modeled the maintenance scheduling at all. We will write false on that branch, and in the first version of our model it will never be taken.

We will now consider the *actions* associated with transitions. They are:

- **Failure** – here we need to place a service request. This is done though the **Main** object by calling its function **requestService()**. The **Main** object is the direct container of the **EquipmentUnit** agent (see Figure 4.13), and there is a function **get_Main()** in the equipment unit that returns the **Main** object. The action of the **Failure** transition, therefore, is `get_Main().requestService( this );`. "this" is a reference to "self": the equipment unit provides it as an argument in the function call to identify who requests the service. That reference will then be placed in the request queue in **Main**.
- **SCArrivedForRepair** and **SCArrivedForMtce** – we need to remember the service crew in a local variable, so the action is `serviceCrew = msg;`. Here, **msg** is the message that triggered the transition.
- **FinishReplacement**, **FinishMaintenance**, and **MaintenanceNotDue** – all these transitions should release the service crew, so they have identical actions:
  ```
  send( "FINISHED", serviceCrew );
  serviceCrew = null;
  ```

If you are observant, you may have noticed that these three transitions are the only transitions entering the **Working** state. So, each time the equipment enters the **Working** state, the same action gets executed. It is tempting to write the code releasing the service crew *just once* in the entry action of the **Working** state instead of writing it three times. However, the entry action will also be executed upon the statechart initialization when there is no service crew to release. Of course, you can test if **serviceCrew** in not null, etc., but to keep the design clean we will leave the code in the transitions; who knows what other transitions we may add to the statechart later that are not necessarily associated with the service crew release.

The last thing we would like to discuss before we switch to the **ServiceCrew** agent is the parameters. The six parameters we created are defined in the editor of the **EquipmentUnit** agent; therefore, each equipment unit will have *its own copy* of **RepairTime**, **BaseFailureRate**, etc. Do we really need that? According to our problem

statement, we don't: the parameters are global and apply to all equipment units. Unless we want to have different values in different units, having a copy in each unit is redundant and consumes memory (not an issue in this model, however). We could define the parameters in **Main**, but then the access to them from **EquipmentUnit** would look a bit ugly: **get_Main().RepairTime**. For simpler code writing, we will leave the parameters where they are.

The ServiceCrew agent

This active object class is also declared as an agent. The initial location, however, is not defined by the environment, because we want the service crews to be initially "at home". So, we will explicitly place it there by writing the following **Startup code** of the agent:

```
Point pt = get_Main().home.randomPointInside();
setXY( pt.x, pt.y );
```

Here, **home** is a polyline drawn in the **Main** object. A random point inside that polyline is selected, and the agent is placed there by calling its function **setXY()**.

Another agent property we need to set up is speed. At the bottom of the **Agent** property, page we set **Velocity** to **500 / day()**. Here 500 model length units (one unit corresponds to one pixel in the editor) means 500 miles, which is consistent with the space settings of the environment object, see Figure 4.11. **day()** is the duration of one day in model time units (if the model time unit is an hour, it returns 24; if it is day, it returns 1; and so on).

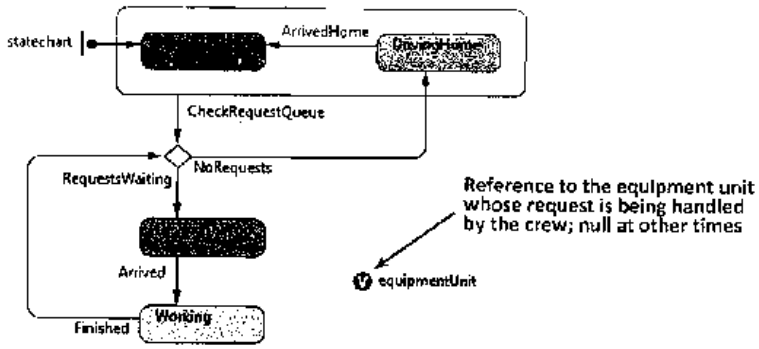


Figure 4.15 AnyLogic statechart of the ServiceCrew agent

The AnyLogic statechart of the service crew is shown in Figure 4.15. The transition triggers and actions are the following:

- **CheckRequestQueue** – triggered by the message "**CHECK REQUEST QUEUE**" sent by the function **requestService** of **Main**.
- **RequestsWaiting** – a branch ("ending" of the two transitions) taken if the condition **get_Main().thereAreRequests()** is true. Action of this branch is
  equipmentUnit = get_Main().getRequest();
  moveTo(equipmentUnit.getX(), equipmentUnit.getY());
  Similarly to the **EquipmentUnit** class, the service crew temporarily remembers the equipment unit being serviced in the variable **equipmentUnit**. **moveTo()** is the built-in function of the agent that starts a straight movement to a given destination point.
- **NoRequests** – the default branch that is taken if the request queue is empty. The action of that branch is same as the Startup code of the agent:
  Point pt = get_Main().home.randomPointInside();
  moveTo(pt.x, pt.y);
  According to the statechart topology, this branch may follow both the transition **Finished** and the transition **CheckRequestQueue**. In the latter case, the service crew is already at home, but it still will make a small move to another location within the home polyline. This, of course, does not happen in reality, but it makes the statechart simpler by not bringing in any considerable error.
- **Arrived** – triggered by the agent's arrival. The action is:
  send(this, equipmentUnit);
  send() is the function of the agent that sends a message to another agent. (Compare this to the call of **receive()** function in **Main**, see Figure 4.13. **Main** is not an agent, so for **Main** that was the only way of delivering a message.) The message is **this** – a reference to service crew itself.
- **ArrivedHome** – triggered by the agent arrival; no action.
- **Finished** – triggered by the message "**FINISHED**" set by the equipment unit, the action code erases the reference to the equipment: **equipmentUnit = null;**.

Animation

We will draw some schematic animation for equipment and service crew (see Figure 4.16) so we see them on the "map" in **Main**. The animation of an agent should be drawn in the agent editor at the coordinate origin so that later on, in the global picture, it appears exactly where the agent is located.

If the animation of the active object was drawn after the object was embedded into a container, it will not appear on the container object diagram until you push the **Create presentation** button in the embedded object properties.

Figure 4.16 Animations of the agents at 300% zoom

In the Main object, we will set up the space where the whole thing will take place. According to the environment settings, the space is 500 x 600 pixels, and one pixel is one mile. The space coordinates are counted from the design-time position of the agent animation, so we will place both the equipment and service crew animations at the same place. We will also draw a rectangle around the space, slightly larger than it (say, 620 x 520 pixels), to visually show the space limits. Also, we will draw a home polyline in the middle of the space, see Figure 4.17.

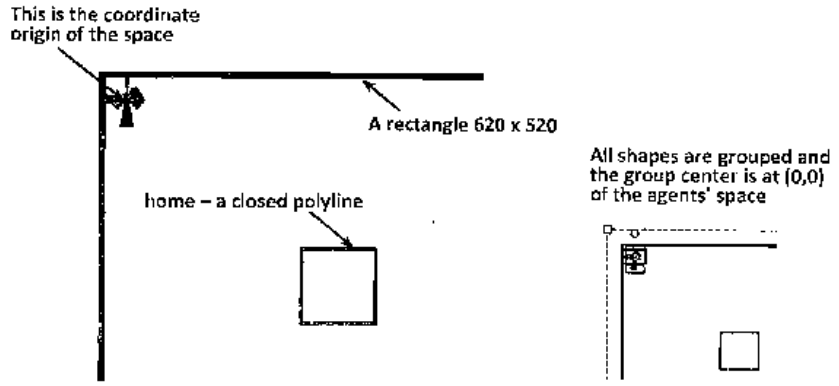

This is the coordinate origin of the space
A rectangle 620 x 520
All shapes are grouped and the group center is at (0,0) of the agents' space
home – a closed polyline

Figure 4.17 The animation of the model assembled in the Main object at 200% zoom

There is one thing we need to fix, however. While the agents' space (0,0) point is where the agents' animations are located, the home polyline's coordinates are counted from the coordinate origin of Main. So, calling `home.randomPointInside()` will return coordinates in a different system. To place the polyline in the same coordinate system as the agents, we should put it in a group with the center at the new coordinate origin. We recommend also adding the framing rectangle and the agent animations in that group, as well.

The first run

Now we can run the model. The top-level animation of the model is displayed in the Main (root) object – the first thing you see after you push the **Run the model** button. You can see how the service crews drive from one equipment unit to another – click the `serviceRequests` queue and inspect its contents.

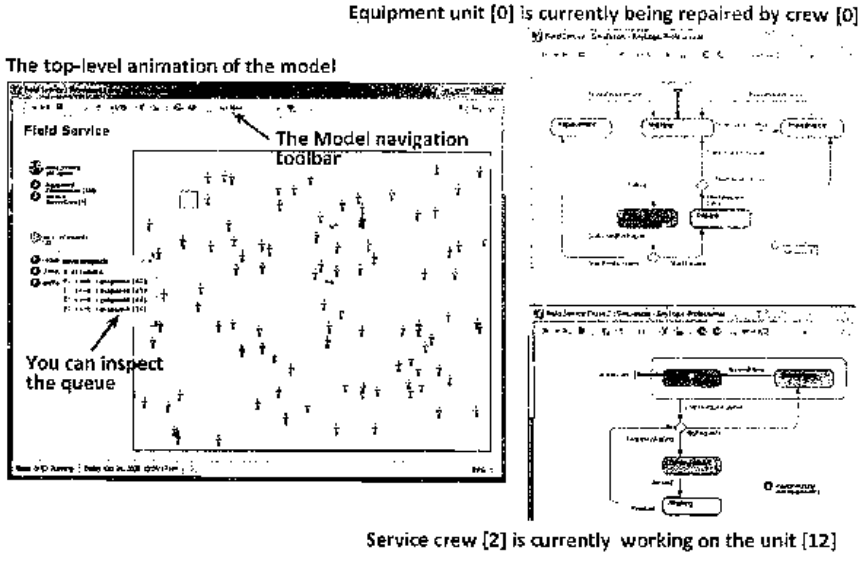

Equipment unit [0] is currently being repaired by crew [0]
The top-level animation of the model
Field Service
The Model navigation toolbar
You can inspect the queue
Service crew [2] is currently working on the unit [12]

Figure 4.18 The first run of the model. Top-level animation and internal view of agents

The great advantage of the object-centric agent-based modeling approach is that we can look inside any agent and see what it is doing, what is its state, variable values, connections, etc, see Figure 4.18. The **Model navigation** section of the AnyLogic runtime toolbar can take you to any object in the model at any depth of the hierarchy.

Discussion and next steps

The main goal of the first runs of the model is to test it. The animation we created may not be needed by the end client, but it greatly helps to verify if the model is working as planned. All main parts of the model are in place now; however, the model at this stage is not yet able to answer our questions. Why not?

- We are not collecting any statistics in our model. We need to know the availability of equipment at any time, the utilization of the service, and many other things.
- Money still is to be added throughout the model. We need to calculate the revenue brought by the equipment and the cost associated with maintaining it.
- Some functionality is still missing (and some parts of the model have not yet been tested); we have not yet added the maintenance of the equipment and

dependency of the failure rate on the equipment age and the timeliness of maintenance.

- We have not programmed the ability to experiment with either the number of service crews and the replacement policy.
- The visual impact of the model can be improved. For example, the color of the equipment unit animation can reflect its state; then, we can always visually assess the percent of working and failed equipment.

We will address these points in the next design phases.

4.4. Phase 3. The missing functionality

Maintenance, age, and failure rate

We will first focus on the missing functionality. According to our problem statement, maintenance is due every 90 days (the **MaintenancePeriod** parameter). When done on time, it decreases the probability of failure. Also, the age of equipment is important: the older the equipment, the higher the failure rate.

How do we discover the time that maintenance is due, and the age? The equipment statechart does not keep that information; when maintenance is done, it just comes to the **Working** state.

The easiest way of always be able to find out the time since a certain event is to *remember the time of the event occurrence in a variable*.

So, we will add two variables to the **EquipmentUnit** agent: **TimeLastMaintenance** and **TimeLastReplacement**. The type of these variables will be **double** because model time in AnyLogic has type **double**. What should the initial values of the variables be? The problem statement says nothing about the initial condition of the equipment, so let us assume that:

- No maintenance is overdue, and the time to the next maintenance is distributed uniformly across the equipment units from 0 to the maintenance period.
- The age is also distributed uniformly from 0 (new) to 3 maintenance periods (moderately old).

Given that the model time at the beginning of the simulation is 0 (the *calendar date* at time 0 can be anything we want), the initial values of the variables will be:

- **TimeLastMaintenance** is initialized as **uniform(-MaintenancePeriod, 0)**
- **TimeLastReplacement** is initially equal to **uniform(-3*MaintenancePeriod, 0)**

We will also write two useful auxiliary functions calculating age and time since last maintenance. **age()** is calculated as **time() – TimeLastReplacement** (in AnyLogic, **time()** is the current model time). Similarly, **timeSinceMaintenance()** returns **time() - TimeLastMaintenance**.

Both functions will return time intervals measured in the time units of the model (see Section 16.1) and it makes sense to set the time units to **days** (this is done in the properties of the model – the top level item in the **Projects** tree).

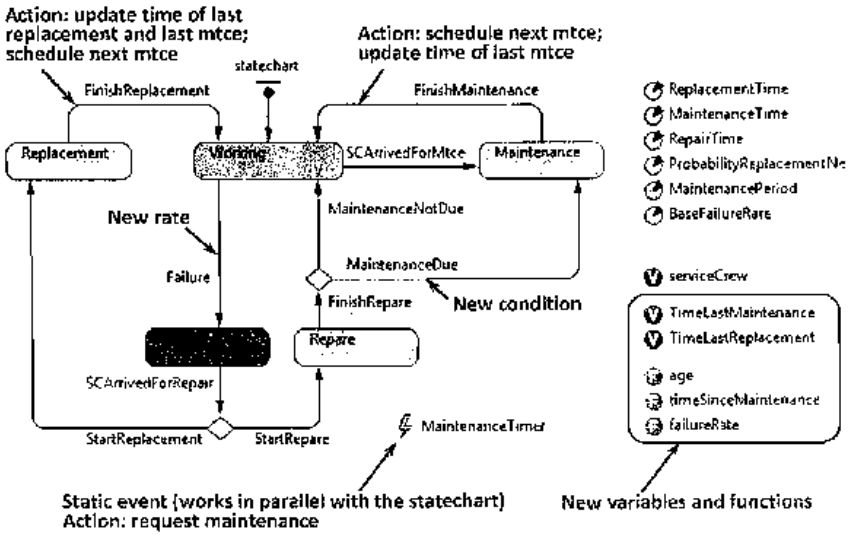


Figure 4.19 Changes in the EquipmentUnit agent: maintenance, age, and failure rate

Since now we know the age and the maintenance state of the equipment, we can properly calculate the failure rate, which depends on both things (as you may remember, in the first version of the model we just used the base failure rate). We were not given the exact dependency formula or any data to fit, so we will assume some simple dependency. For example:

- If maintenance is overdue, the increase of the failure rate is proportional to the overdue period divided by the maintenance period. For example, if the maintenance has not been done for 180 days (it is 90 days overdue), the failure rate will be multiplied by 2.
- Similarly, the equipment older than 3 maintenance periods has the failure rate increased by the age divided by 3 maintenance periods.

This is the code of the **failureRate** function:

```
double mtceoverduefactor = max( 1, timeSinceMaintenance() / MaintenancePeriod );
double agefactor = max( 1, age() / ( 3 * MaintenancePeriod ) );
return BaseFailureRate * mtceoverduefactor * agefactor;
```

Now we can use this function in the rate of the Failure transition. Also, we can now write the proper condition of the transition branch MaintenanceDue; it will be timeSinceMaintenance() > MaintenancePeriod. The changes made to the model in this and the next section are highlighted in Figure 4.19.

Scheduling maintenance. Handling requests of two types

Our model is still missing the maintenance scheduling. Which object in the model should take care of that? In the real world, it is most likely the service system that keeps the log of completed maintenances and schedules the next ones. We could literally mirror this in the model by adding an array of dynamic events (see Section 8.3), one for each equipment unit, at the Main level. However, it would be more natural and elegant to have a static event (see Section 8.2) in each equipment unit. The event will act like a timer.

The equipment unit has a statechart that defines the main behavior pattern. Now we are adding a small primitive activity (an event) that will go on *in parallel* to the main activity and interact with it, see the message sequence diagram in Figure 4.20.

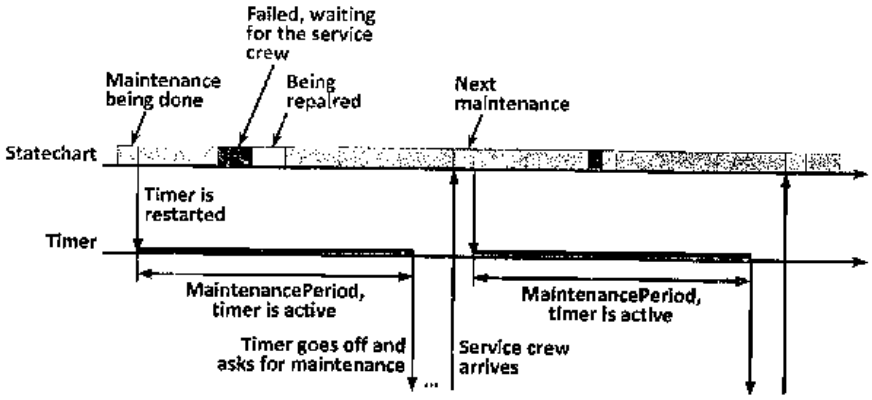
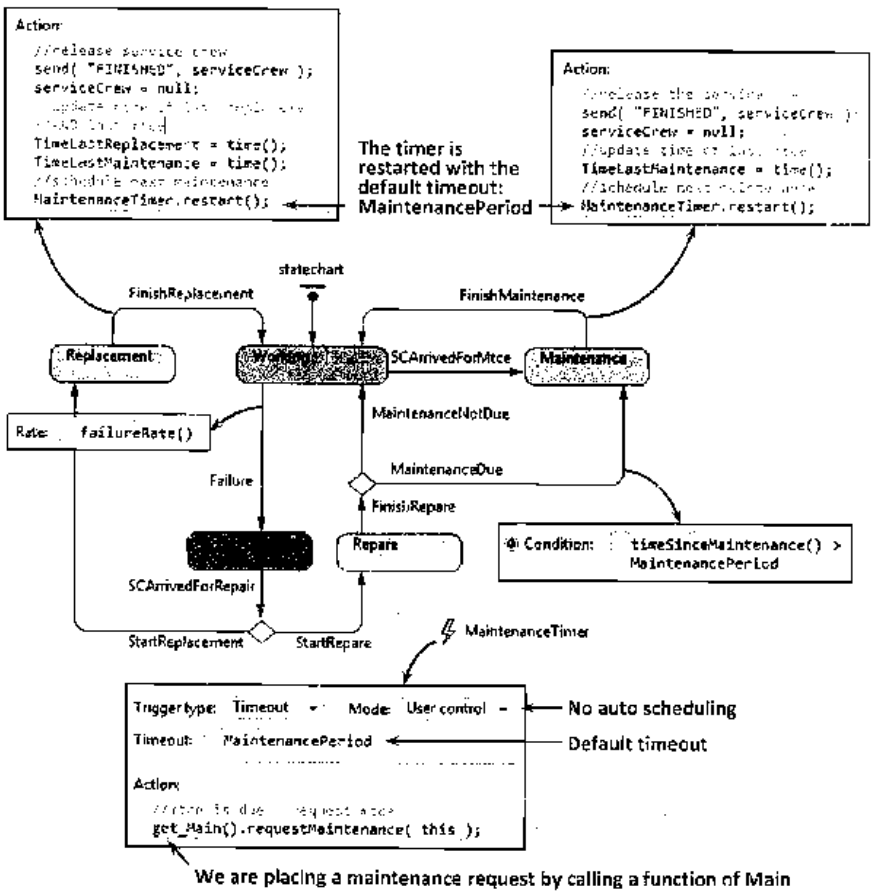


Figure 4.20 Two parallel activities in the EquipmentUnit agent: statechart and timer

We will call the event MaintenanceTimer. The timer will be restarted each time the maintenance is done for the MaintenancePeriod. And we need to initialize the timer at the beginning of the simulation according to the initial value of the TimeLastMaintenance variable. The corresponding settings of the model are shown in Figure 4.21.



Startup code of the EquipmentUnit agent

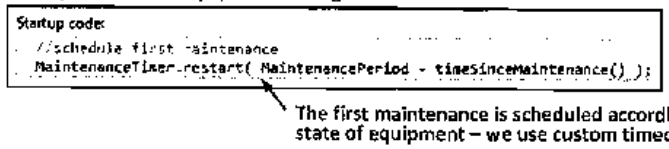

Startup code:
//schedule first maintenance
MaintenanceTimer.restart(MaintenancePeriod - timeSinceMaintenance());
The first maintenance is scheduled according to the initial state of equipment – we use custom timeout

Figure 4.21 Properties of objects related to maintenance scheduling

When the maintenance timer goes off, the equipment unit will request a service crew to come and perform the maintenance by calling the function **requestMaintenance()** of the **Main** object. That function still is to be implemented; so far, in the **Main** object, we only have one request queue and one function **requestService()** that puts the request there. Should we place the maintenance

requests in the same queue? If we decide to do so (and leave the reference to the equipment unit as the request type), the requests will be indistinguishable and we will not be able to treat them differently: for example, to implement priorities. There are a couple options. Option one is to create a special type for a request that will include the reference to the equipment unit, the type of request (repair or maintenance), and maybe more information, such as timestamp. Option two is to have two different queues for equipment units that need repair and those that need maintenance. For simplicity's sake, we will choose the second option.

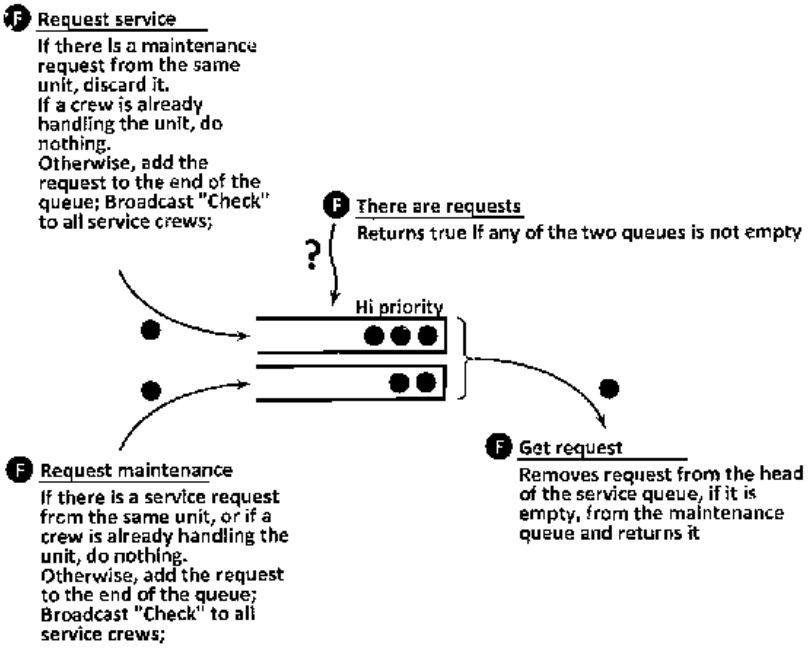

Request service
If there is a maintenance request from the same unit, discard it.
If a crew is already handling the unit, do nothing.
Otherwise, add the request to the end of the queue; Broadcast "Check" to all service crews;
There are requests
Returns true if any of the two queues is not empty
?
Hi priority
Request maintenance
If there is a service request from the same unit, or if a crew is already handling the unit, do nothing.
Otherwise, add the request to the end of the queue;
Broadcast "Check" to all service crews;
Get request
Removes request from the head of the service queue, if it is empty, from the maintenance queue and returns it

Figure 4.22 The "service dispatcher" in Main that handles two types of requests

The updated implementation of the "service dispatcher" is shown in Figure 4.22. We will assume that service requests are treated as high priorities because the failed equipment stops bringing revenue. The maintenance requests are served only if there is no failed equipment.

The equipment unit, however, can generate two requests in parallel. For example, it can request maintenance and, while waiting for the service crew, it can fail and ask for repair. Or, the maintenance timer can go off while the equipment is being repaired. These situations need special treatment, because we do not want two service crews to arrive at the same equipment unit. Therefore, before adding a

request to a queue, we are checking if the there was another request from the same unit. The code of the corresponding functions of **Main** is shown in Figure 4.23.

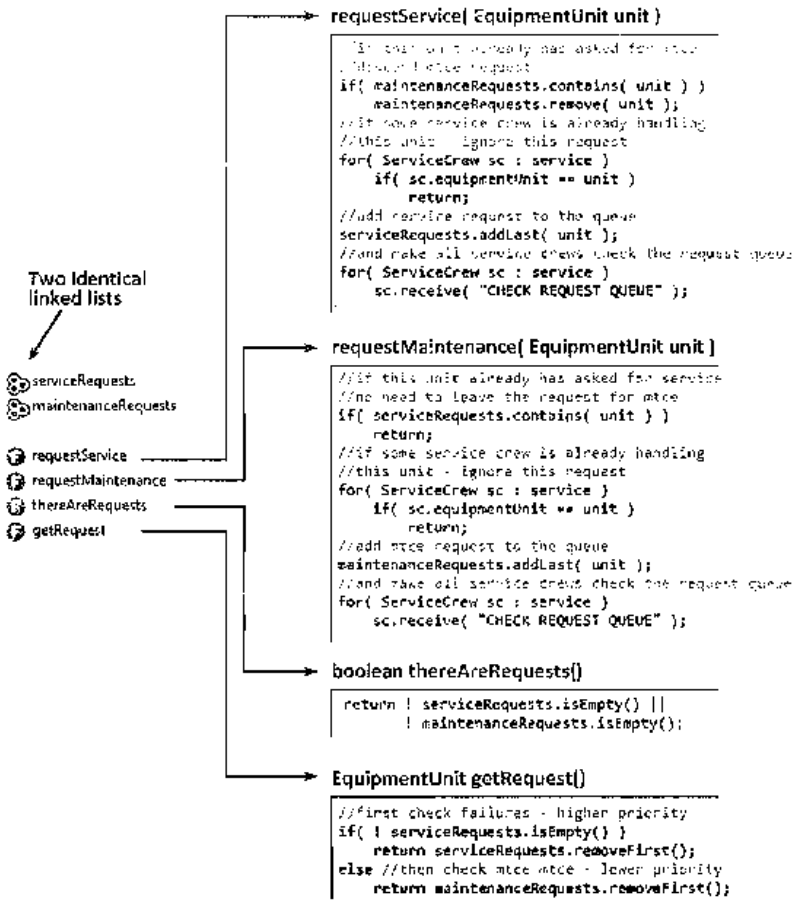

requestService(EquipmentUnit unit)
//if this unit already has asked for mtce
//then ! mtce request
if(maintenanceRequests.contains(unit))
maintenanceRequests.remove(unit);
//if some service crew is already handling
//this unit - ignore this request
for(ServiceCrew sc : service)
if(sc.equipmentUnit == unit)
return;
//add service request to the queue
serviceRequests.addLast(unit);
//and make all service crews check the request queue
for(ServiceCrew sc : service)
sc.receive("CHECK REQUEST QUEUE");
Two Identical linked lists
serviceRequests
maintenanceRequests
requestService
requestMaintenance
thereAreRequests
getRequest
requestMaintenance(EquipmentUnit unit)
//if this unit already has asked for service
//no need to leave the request for mtce
if(serviceRequests.contains(unit))
return;
//if some service crew is already handling
//this unit - ignore this request
for(ServiceCrew sc : service)
if(sc.equipmentUnit == unit)
return;
//add mtce request to the queue
maintenanceRequests.addLast(unit);
//and make all service crews check the request queue
for(ServiceCrew sc : service)
sc.receive("CHECK REQUEST QUEUE");
boolean thereAreRequests()
return ! serviceRequests.isEmpty() ||
! maintenanceRequests.isEmpty();
EquipmentUnit getRequest()
//first check failures - higher priority
if(! serviceRequests.isEmpty())
return serviceRequests.removeFirst();
else //then check mtce mtce - lower priority
return maintenanceRequests.removeFirst();

Figure 4.23 The updated implementation of the service system in Main

Now the model functionality is completed and we can run the model again to check if equipment maintenance is done. You can see the number of maintenance requests in the queue in the **Main** object, the time to the next maintenance nearby the **MaintenanceTimer** in **EquipmentUnit**, and the equipment statechart periodically entering the **Maintenance** state.

Discussion. Code in the model

As you can see, we have used some code. The code was needed for:

- Calculation of the equipment failure rate.
- Interaction of the statechart and the timer in the **EquipmentUnit** agent.
- Communication of the "service dispatcher" in **Main** and the service crews.
- Queue management in **Main**.

In the case of the failure rate, we had a complex relationship between variables originally described in an *algorithmic manner*: "if the equipment is older X then the failure rate increases by the age divided by X". Such a relationship is naturally implemented by code, and you will find code like this in simulation models of any kind. The code includes declaration of auxiliary variables and arithmetic calculations, sometimes conditional. For example:

- **double agefactor = max(1, age() / (3 * MaintenancePeriod));** – this code line declares a local variable **agefactor**, calculates the expression **age() / (3*MaintenancePeriod)**, and assigns its value to the variable if it is greater than 1; otherwise, it assigns 1.

The queue management code has a similar nature. The rules like "if a request is being added to the service queue, delete a request from the same equipment unit in the maintenance queue if there is any" are also best expressed in code. As long as we use linked lists for queues, we use the methods of Java collections (see Section 10.6). For example:

- **boolean contains(*<element type>* element)** – returns true if the collection contains the **element** and false otherwise.
- **remove(*<element type>* element)** – removes the **element** from the collection.
- **addLast(*<element type>* element)** – adds the **element** to the end of the linked list.

The other two cases (interaction of the statechart and the timer and communication of **Main** and **ServiceCrew**) are typical for agent based modeling.

In agent based models, code is frequently used to link things that are not linked graphically. In particular, it is used to implement agent-to-agent and agent-to-environment communication, and to co-ordinate different activities within a single agent.

These are the examples of such code:

- The statement **MaintenanceTimer.restart()** written in the action code of a statechart transition addresses the static event **MaintenanceTimer** located in the same agent.

- The statement get_Main().requestMaintenance() in the action code of the MaintenanceTimer located in the EquipmentUnit agent reaches out to the equipment's container object Main and calls its function requestMaintenence().
- The loop statement in Main

```
for( ServiceCrew sc : service )
  if( sc.equipmentUnit == unit )
    return;
```

iterates through all ServiceCrew agents embedded in Main (service is the name of the service crew collection), checks if the variable equipmentUnit in the service crew equals a given unit, and, if yes, exits the loop and the function where the loop is located.

The code patterns we considered are very typical and cover about 90% of the modeler's needs. For further information on how to write code and what can be done by code, we refer you to Chapter 10, "Java for AnyLogic users".

4.5. Phase 4. Model output. Statistics. Cost and revenue calculation

Now that the functionality is in place, we can work on the model output. So far, animation is the only output the model generates. In this design phase, we will add revenue and cost calculations throughout the model, and collect and visualize statistics. These are the output metrics we are interested in:

- Equipment availability
- Service crew utilization
- Cost of the service system
- Revenue

Equipment availability and service crew utilization

Both the equipment and the service crews are agents, so the task of calculating availability and utilization is the task of obtaining percentages and average numbers across the collections of agents. In AnyLogic, such statistics can be defined on the **Statistics** page of the agent collection properties, see Figure 4.24.

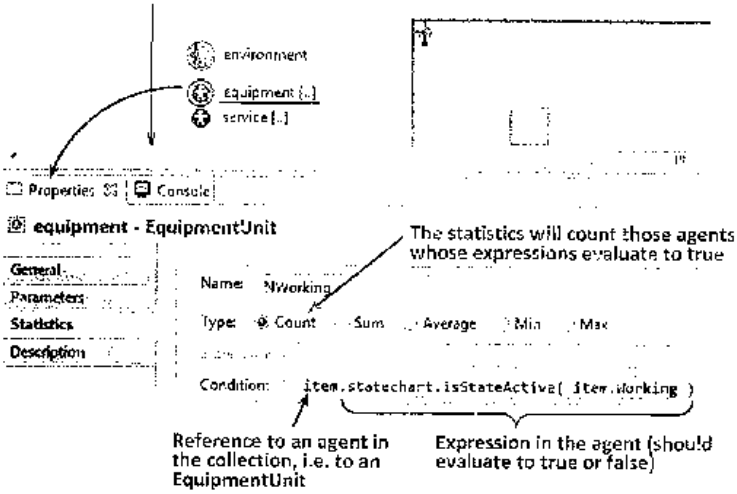


Figure 4.24 Statistics defined in the collection of the equipment units

In those statistics we are counting the working equipment units, i.e. the units where the current statechart state is **Working**. The expression in **Condition**, filed, looks a bit unusual: the "natural" or "expected" form of the expression would be something like **statechart.isStateActive(Working)**. To understand why you should write the expression this way consider the Java code generated by AnyLogic for the statistics (the user's part of the code is highlighted):

```
int count = 0;
for( EquipmentUnit item : equipment ) {
  if( item.statechart.isStateActive( item.Working ) )
    count++;
}
```

We are currently in the **Main** object and *outside* the **EquipmentUnit** object. Therefore, according to Java rules (see Section 10.9), every function or variable inside **EquipmentUnit** must be *prefixed by the reference to the object*. The local variable **item** in the loop is the reference to the equipment unit, so it is put in front of the statechart and the state names.

You may also wonder why we did not select the statistics type **Average**. Average calculates the average of a numeric value across the collection. For example, if we were calculating the average age of the equipment, we would use type **Average**. The working / not working status of the equipment is a **boolean** value, so we are counting the working units.

The result of the statistics definition is the function **NWorking()** in the Main object that returns the number of working units at the time it is called. Similarly, we will define three other statistics functions:

- **NOnService()** calculating the number of equipment units being repaired or replaced with a slightly more complex condition:
  item.statechart.isStateActive(item.Repare) ||
  item.statechart.isStateActive(item.Replacement)
- **NOnMaintenance()** with the condition
  item.statechart.isStateActive(item.Maintenance)
- **NFailed()** with **item.statechart.isStateActive(item.Failed)**

So far, we have only prepared the tools to collect data; no data has actually been collected. We will now add a time stack chart to view the equipment availability.

The stack chart settings are shown in Figure 4.25. There are four stripes in the chart, corresponding to the four different statistics we have defined. The colors of the stripes are the same as the colors of the equipment statechart states.

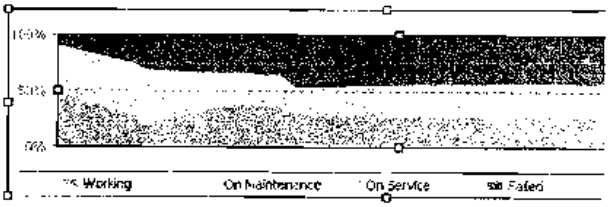
In the first version, the chart will display a time interval of one year (see the **Time window** property). Once per day (**Recurrence time** setting) it will evaluate all four data items (call the four statistics functions), and will internally keep a dataset of 365 latest samples (**Display up to** setting).

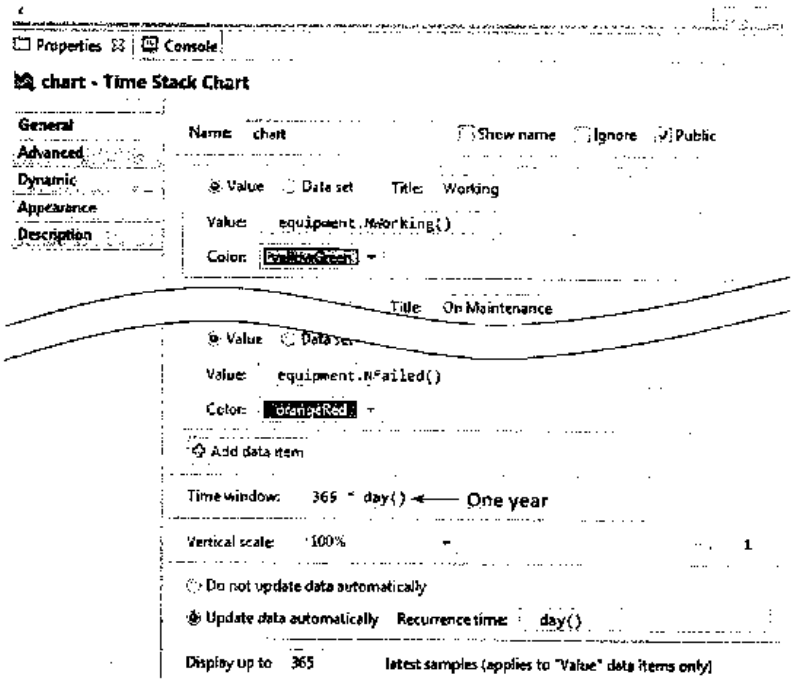
As long as the total number of equipment units is constant and we are interested in the percent of working units, the **Vertical scale** is set to 100%. This means that the total height of the four stripes will always equal the full height of the chart.

In most cases, it makes sense to separate the animation and output screens. Moreover, the amount of output data may require more than one screen. Use AnyLogic view areas to markup the screens and hyperlinks to switch between them (see Example 13.10: "Hyper link menu to navigate between view areas").

The runtime picture of the chart after one year is shown at the bottom of Figure 4.25. Not surprisingly, the percent of time spent on service and maintenance times is minor compared to the time equipment is up and running. But the time the equipment is failed and waits for the service crew (the red stripe) is significant.

This chart is good as a first iteration of our model output design. However, the chosen time window of one year is too short to see the effect of a service policy change, given the order of magnitude of the maintenance period and the impact of age on the failure rate. The sample period (once per day) may, on the contrary, be too long compared to the maintenance and repair times. The last problem with the chart is the noise that hides the possible "tides" in the system.





The chart at runtime after one year has been simulated

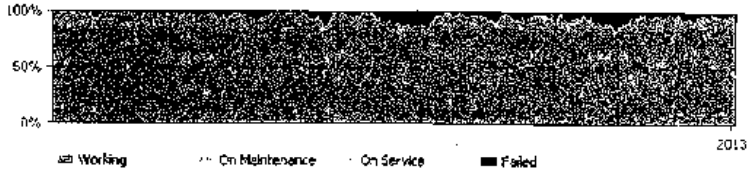


Figure 4.25 Time stack chart for equipment availability. Design time settings and runtime view

We will now modify the statistics collection and visualization to handle these issues. We will:

- Use a higher sampling rate (once per hour) to better monitor the state of the equipment.
- Accumulate the *time averages* of the equipment states during a year.
- Display annual averages in the same stack chart to see the 50 years' interval.

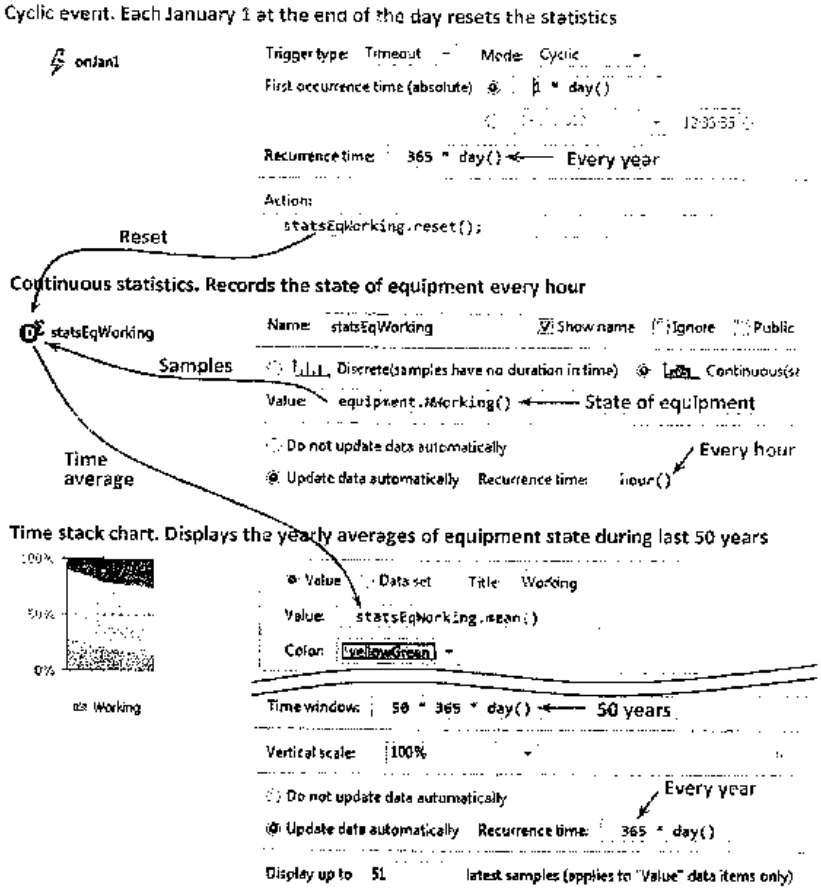


Figure 4.26 Statistics defined in the collection of the equipment units

The element of AnyLogic that is capable of calculating time averages is called continuous statistics. To create continuous statistics, you should drag the **Statistics** object from the **Analysis** palette and select **Continuous (samples have duration in time)** in its properties. Every hour, we will add the number of working equipment units to

the statistics. By the end of the year (at the end of the last day of a year) the statistics will have the full history of the equipment state, and we will add the time average (the function mean() of the statistics) to the stack chart, see Figure 4.26. The day after, we will reset the statistics so that it starts collecting data for the next year. This is done by the cyclic event onJan1 that is set up to occur at the end of the 1st day, 366th day, etc.

In the same manner, we will define the statistics for the on maintenance, on service, and failed states of equipment, and for the idle, driving, and working states of the service crews. The two stack charts with the results of 50 simulated years with default parameter values are shown in Figure 4.27. These charts (along with the financial ones that we will add in the next section) will be the basis for our policy analysis.

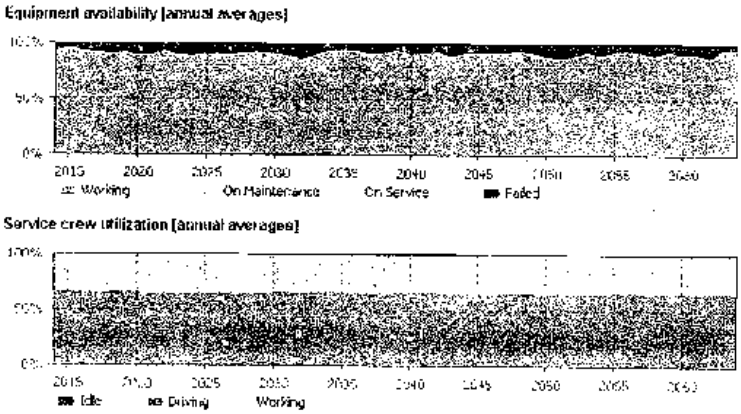


Figure 4.27 Annual averages of equipment and service states during a 50 years period

Cost and revenue

How do we calculate the cost of our service system? There are two components of cost:

- Daily cost of "employing" the service crews, and
- Per-operation cost of maintenance, repair, and replacement.

The first part of the cost equals (the number of service crews) x (daily cost of a crew) x (365). Potentially, we may want to vary the number of service crews during a year, so we will use the time average number (yet another continuous statistics will be needed).

As for the cost of operations being performed, the information is currently not recorded anywhere. We will create a variable **WorkCost** at the top level in **Main** and

increment it each time a maintenance, repair, or replacement operation is performed. In addition, we will create the cost and revenue parameters in the **Main** object. The changes made to **Main** are shown in Figure 4.28.

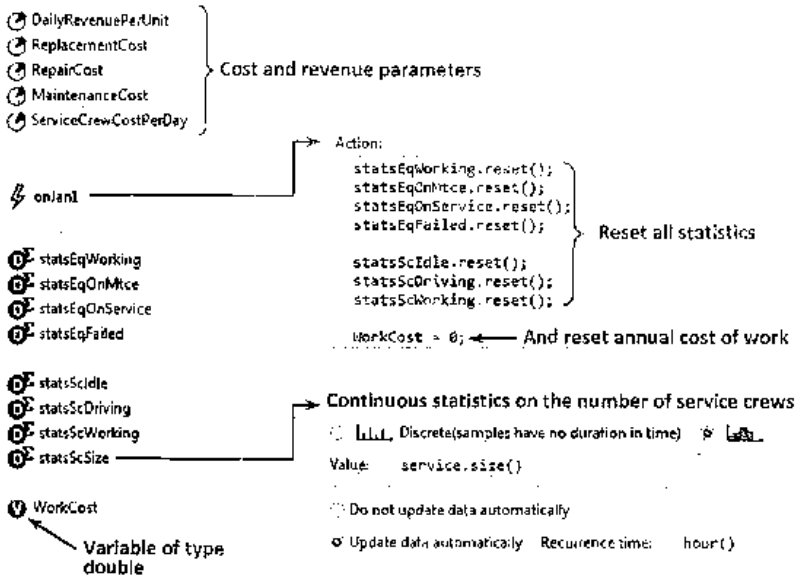


Figure 4.28 Elements in Main related to cost and revenue calculations

In the **EquipmentUnit** agent we will include the increments of the **WorkCost** variable at the end of each operation. The corresponding actions are added to the transitions **FinishMaintenance, FinishRepair**, and **FinishReplacement**, see Figure 4.29. As long as both the cost parameters and the variable **WorkCost** are located in **Main**, we need to use the prefix **get_Main()** to access them from the equipment unit.

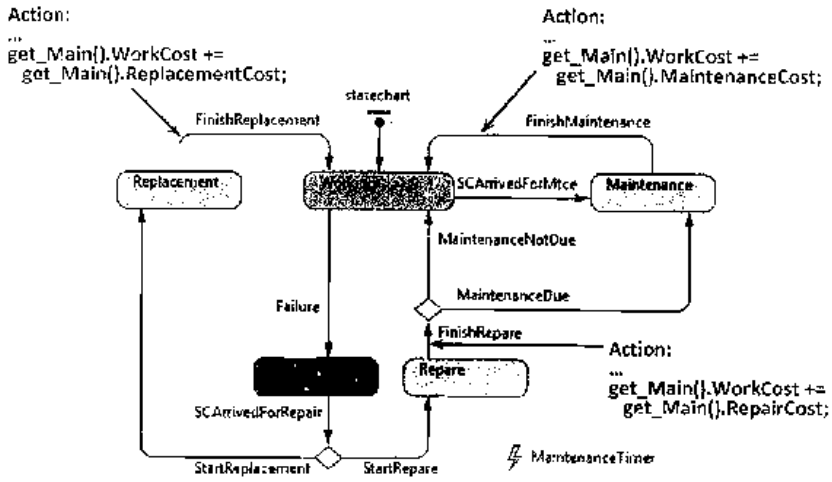


Figure 4.29 EquipmentUnit statechart updated to calculate the cost of operations

Revenue calculation is easy. A working equipment unit is bringing `DailyRevenuePerUnit` dollars per day. Therefore, the annual revenue can be calculated as `statsEqWorking.mean() * DailyRevenuePerUnit * 365`.

Finally, we will create a chart of annual cost, revenue, and profit. We will use a time plot, see Figure 4.30. The time window, recurrence, and history size settings are the same as in the time stack charts we created before. To keep the numbers smaller, we will display them in thousands dollars (all values are divided by 1000).

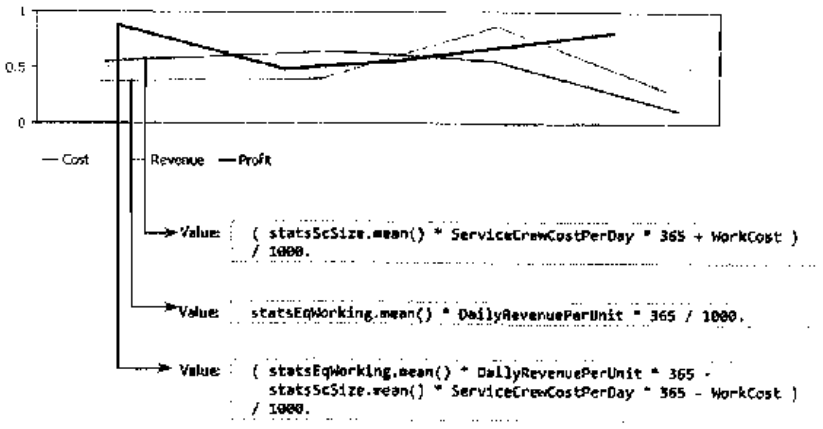

1
0.5
0
Cost
Revenue
Profit
Value: (statsScSize.mean() * ServiceCrewCostPerDay * 365 + WorkCost) / 1000.
Value: statsEqWorking.mean() * DailyRevenuePerUnit * 365 / 1000.
Value: (statsEqWorking.mean() * DailyRevenuePerUnit * 365 - statsScSize.mean() * ServiceCrewCostPerDay * 365 - WorkCost) / 1000.

Figure 4.30 Cost and revenue chart

We have completed the measurements, statistics, and output visualization for our model. The resulting output screen is shown in Figure 4.31.

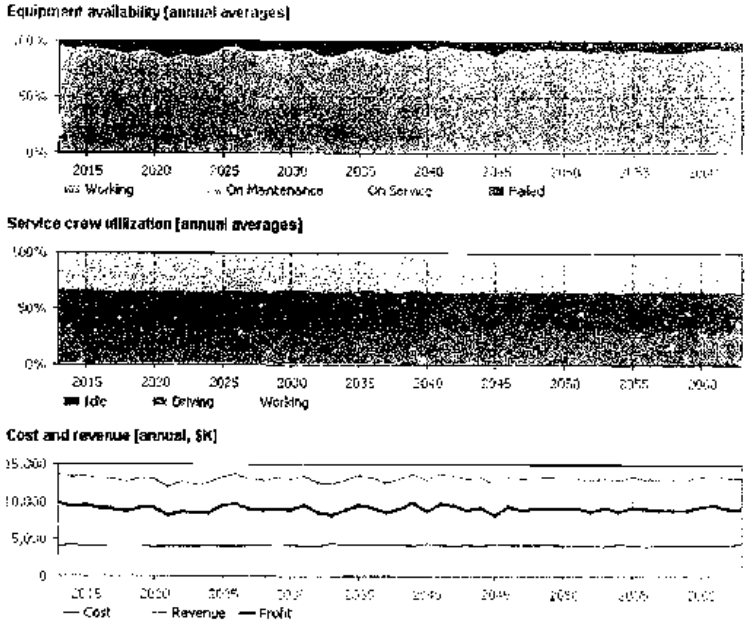


Figure 4.31 The model output screen with cost and revenue. 50 years simulated

4.6. Phase 5. Control panel. Running the flight simulator

We will now convert our model into a flight simulator where parameters and policies can be varied on the fly and the results are displayed immediately. The model user will be able to vary two things:

- The number of service crews, and
- The replacement policy, namely: a) replace only failed equipment that cannot be repaired, or b) also replace equipment after N maintenance periods, even if it is still working

Design of control panel

We will create a simple control panel with two groups of radio buttons and one edit box, see Figure 4.32. We will need three more parameters in the model:

- **ServiceCapacity** –the number of service crews, integer type, initially 3.

- **ReplaceOldEquipment** – boolean, initially false, if true, the equipment is replaced after **MtcePeriodToReplace** maintenance periods.
- **MtcePeriodToReplace** – integer, initially 5, the maximum age of equipment if the **ReplaceOldEquipment** policy applies.

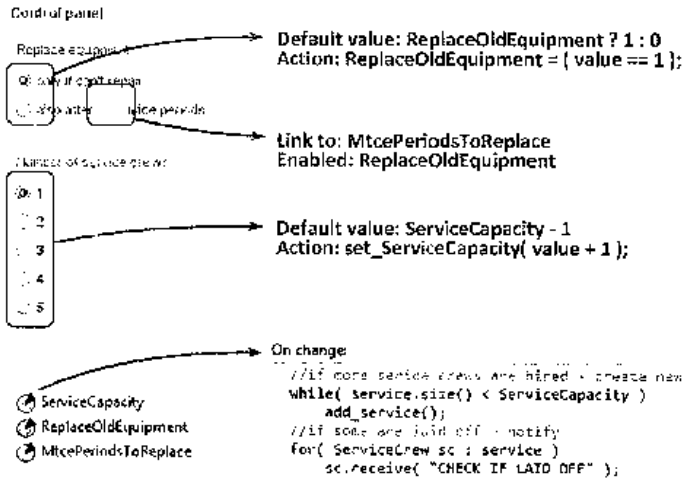


Figure 4.32 The model control panel and related parameters

The value of a radio button control is a 0-based integer. Therefore, if the user selects **Replace equipment only if can't repair**, the value is 0, and the boolean parameter **ReplaceOldEquipment** is set to **(value == 1)**, which evaluates to **false**. The edit box is linked to an integer parameter, so it will automatically accept only an integer input.

Now we need to set up the model to properly react to the policy changes made on the fly.

Changing the number of service crews

In AnyLogic, agents can be added or deleted to or from a collection by calling the functions **add_*<collection_name>*()** and **remove_*<collection name>*(agent)**. To add a new service crew to our service system, we can simply call **add_service()**. However, we *cannot delete a service crew at an arbitrary moment of time*: the crew can be handling a service request and, if the crew is deleted in the middle of say, a repair operation, the operation will not finish correctly. For example, the message **"FINISHED"** will be sent to a non-existing agent. To delete a service crew correctly, we first need to make sure it is idle.

Here you can see the difference with discrete event (process-based) modeling. In a discrete event model you typically can reduce the number of resources in a pool at any time and, if the resource being deleted is busy, it will follow some default policy; for example, finish the work and then disappear. In the agent based model, we are implementing this policy explicitly.

Our implementation will be the following. Each agent in a collection has an index (from 0 to the size of the collection - 1), and we will use that index to check if the service crew is still employed. Each time a service crew finishes work with equipment, it will check if its index is greater or equal than the parameter **ServiceCapacity**. If yes, it will immediately delete itself. And for those service crews that are idle or driving home, we will send a special notification to make check the same condition.

Part of this scheme is implemented in **Main** in the **On change** code of the parameter **ServiceCapacity**, see Figure 4.32. The second part is implemented in the updated **ServiceCrew** agent, see Figure 4.33. The decision diamond on the left has two entries: the **Finished** transition and the **CheckIfLaidOff** transition, and two outgoing branches: one leads back to the normal operation, and another to the final state, where the agent deletes itself (**this** is the Java reference to self).

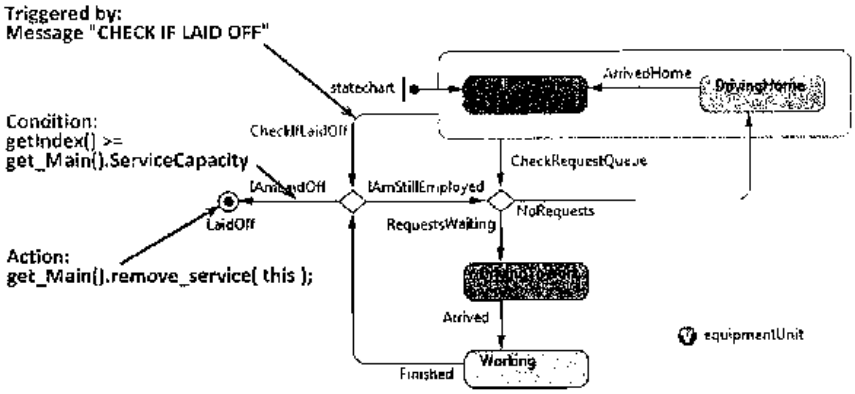


Figure 4.33 The ServiceCrew statechart updated to react to the service capacity changes

Equipment replacement policy

We need to implement the optional replacement of the equipment after a given number of maintenance periods. We already have **ReplaceOldEquipment** and **MtcePeriodsTpReplace** parameters in **Main**. The rest can be naturally done in the

EquipmentUnit agent. Each time the service crew visits the equipment unit, we will check if the policy is active; if yes, we will compare the age of the equipment with MtcePeriodsToReplace * MaintenancePeriod. If the age is greater, the crew will perform the "planned" replacement of the working equipment.

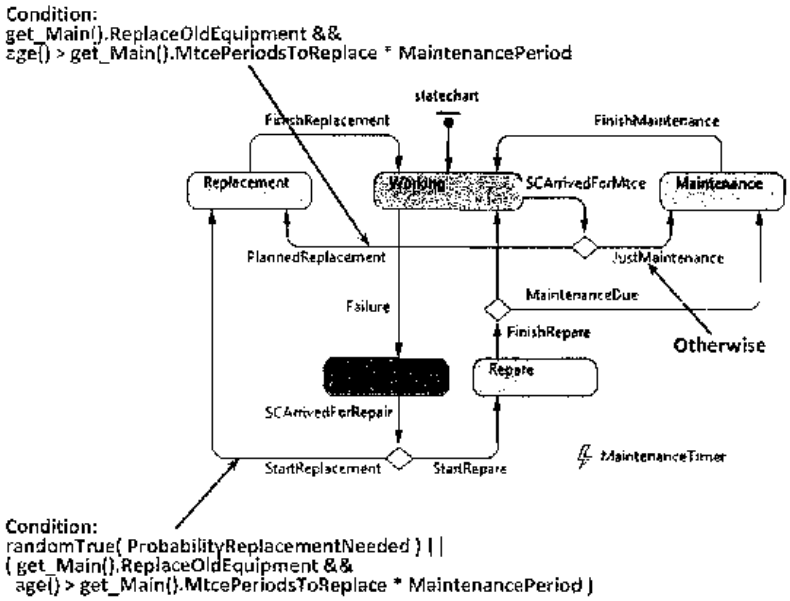


Figure 4.34 The EquipmentUnit statechart implementing the optional replacement policy

We added another decision diamond after the transition **SCArrivedForMtce** ("service crew arrived for maintenance"). If a planned replacement is recommended and due, the statechart proceeds directly to the **Replacement** state. We also modified the condition of the **StartRepare** branch. Now, if the service crew comes to fix the failed equipment, it would also check if the planned replacement needs to be done.

The model now is ready for experiments.

Running the flight simulator

In the first experiment, we will vary the number of service crews. The results are shown in Figure 4.35. The profit is virtually the same with four crews as with three, but the equipment availability is considerably higher with four crews. The further increase of the service capacity does not result in a noticeable increase of

availability, but it lowers the profit. Less than three service crews cannot handle the fleet of 100 equipment units.

While, in reality, the system we are modeling will never remain unchanged for 50 years, the 50 years' time window is good for the flight simulator mode as it allows us to compare different solutions on one chart.

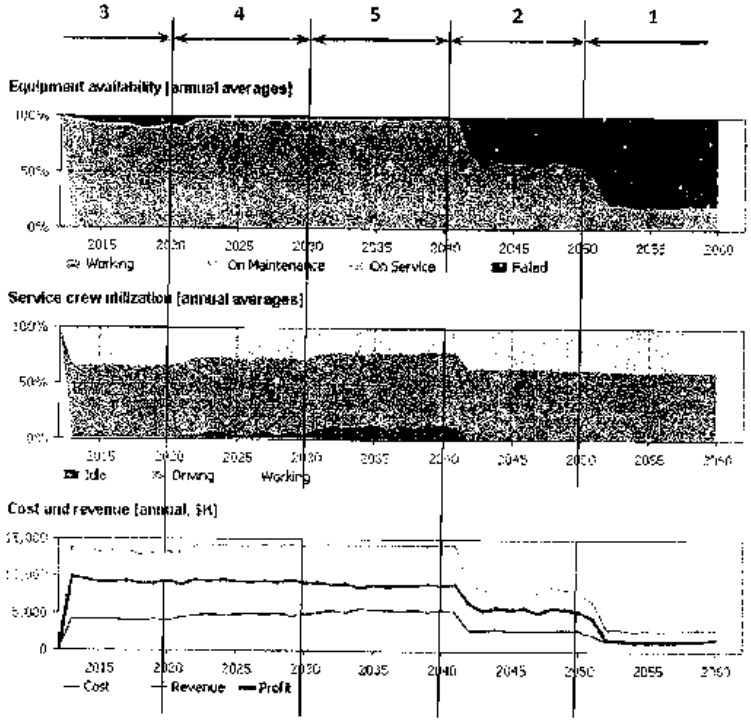


Figure 4.35 Trying different numbers of service crews

In the next run of our flight simulator, we will try three and four service crews and vary the equipment replacement policy. The results (see Figure 4.36) show that three service crews can successfully maintain high equipment availability and, at the same time, bring high profit if old equipment is replaced.

Should we also try two service crews? We can, but even with the three parameters we can vary (the number of service crews, the active/inactive policy, and the maximum allowed age of equipment) it is quite hard to manually explore the parameter space. Our next step will be to submit this task to the automatic optimizer.

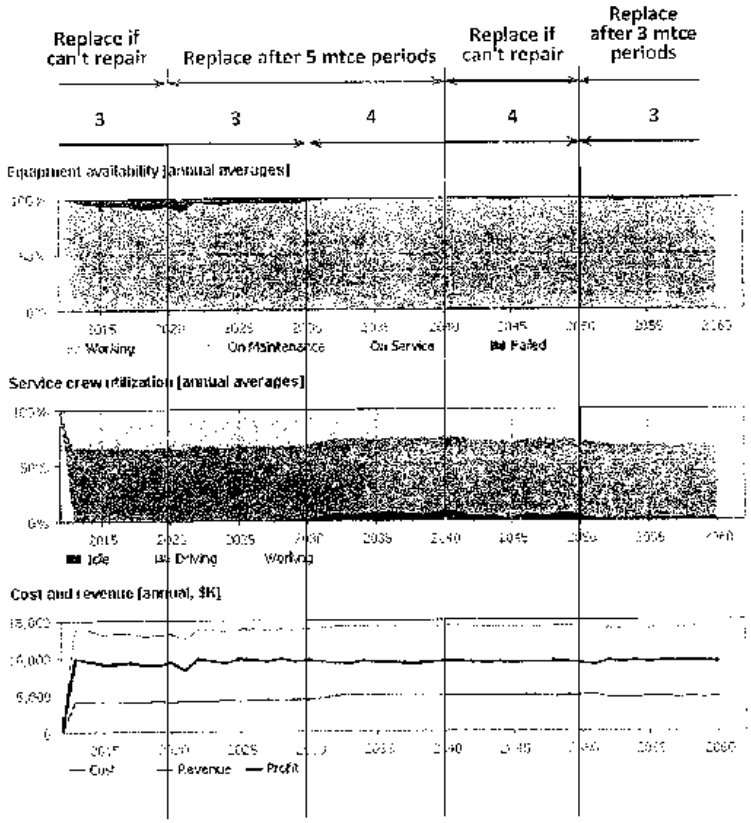


Figure 4.36 Trying 3 and 4 service crews with different replacement policies

4.7. Phase 6. Using the optimizer to find the best solution

We will use the OptQuest™ optimizer that is embedded into AnyLogic. Being a metaheuristic optimizer, OptQuest treats the simulation model as a black box and can work with models of any kind, be they discrete event, agent based, or system dynamics models. It is also capable of working with stochastic models and performing optimization under uncertainty.

Preparing the model for optimization

Before creating the optimization experiment, we need to prepare the model. During optimization, we are not interested in animation and visualization of the statistics. We are, however, interested in performing simulation runs as fast as possible, because there will be many of them. As to the animation graphics and the controls,

we can either delete them or leave them as-is; when not shown on the screen, AnyLogic animation does not consume any computational resources. Charts and statistics elements, however, are set up to evaluate their values with a certain frequency, which slows down the simulation (especially when evaluating a value requires iteration over a collection of agents). Therefore, we will delete:

- All charts.
- All statistics except for **statsEqWorking**, which is needed to calculate the revenue.
- Event **onJan1** that resets the statistics every year.

We will create a new function **profit()** that will be called once at the end of a simulation run and calculate the cumulative profit. This will be our objective function. Figure 4.37 shows what is left.

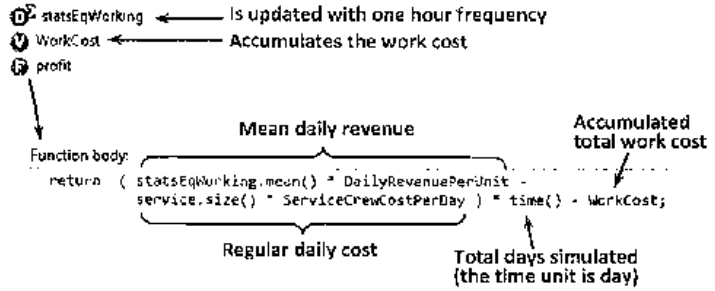


Figure 4.37 The statistics and the objective function needed for optimization

Setting up the optimization experiment

One of the important settings is the duration of a simulation run. The duration should be long enough to let the model work in the regular mode and produce meaningful output. The warm-up period, in our case, is not a big issue: we are initializing the equipment units at randomly distributed age and last maintenance dates, so the initial state of the model differs from a regular state only in the following:

- None of the equipment has failed or is being repaired.
- All service crews are idle and at their home location.
- The service and maintenance request queues are empty.

Given the failure rates and the maintenance period, the model should enter the regular mode in less than a year. We will set the duration of a run to 20 years and will not exclude the warm-up period from the output.

We will ask the optimizer to maximize the cumulative profit evaluated at the end of a simulation run, so we just write root.profit() in the objective function (root is the reference to the top-level object in the model, which in our case is Main).

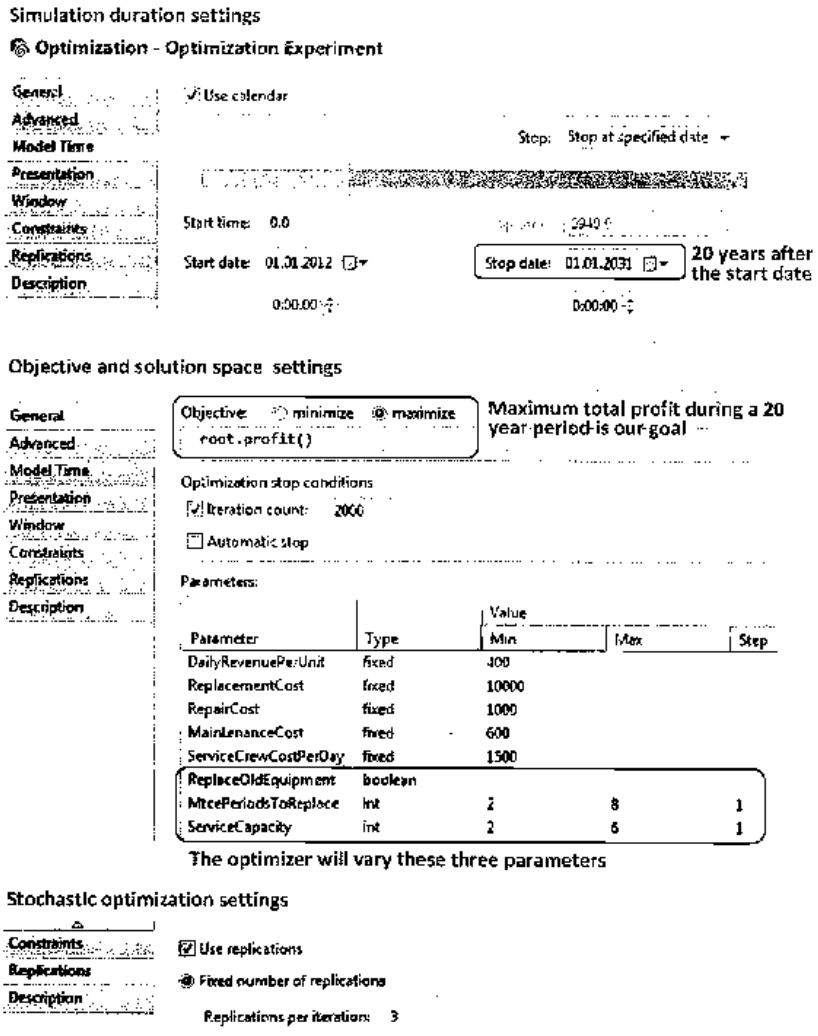


Figure 4.38 Settings of the optimization experiment

The model has eight parameters at the top level Main object and more in the EquipmentUnit agent. The solution space, however, is limited to only three

parameters (all of them are in Main): ServiceCapacity, ReplaceOldEquipment, and MtcePeriodsToReplace. The solution space settings are shown in Figure 4.38.

From the flight simulator runs, we remember that two or fewer service teams are not capable of handling the equipment, and five teams will be underutilized. Hence, the minimum and maximum values of the ServiceCapacity parameter. The range for the MtcePeriodsToReplace [2..8] is chosen both from the experience of the previous runs and from the knowledge of the failure rate function.

The last thing we need to set up is the number of replications. Our model is clearly stochastic (see Section 15.2) and, given that the seed of the random number generator (see Section 15.3) is chosen randomly, each run may produce a different output. Therefore, will perform multiple runs for each set of the input parameters (multiple replications) and use the distribution of the simulation output instead of a single value. A large number of replications obviously slows down the optimization, so we will set it to three.

The methodology for dealing with the model warm-up and for choosing the number of replications is outside the scope of this chapter and is well described, for example, in (Kelton D., Sadowski R. & Sturrock D., 2004).

Optimization run

Now we can run the optimization. On the author's machine, the optimization was completed in about two minutes and produced the results shown in Figure 4.39. The default user interface of the experiment was slightly modified. The chart on the right shows the progress of the optimization. Each dot in the chart corresponds to an iteration performed (each iteration has three runs, according to our replication settings). The vertical axis is the value of the objective function.

The best solution found by the optimizer is this:

The solution

- There are 3 service teams.
- We do replace old equipment, even if it works.
- The maximum allowed equipment age is 4 maintenance periods, i.e. one year.

Under these conditions, the estimated total 20-year profit is $180,758,172, which makes an average annual profit of $9,037,909 (the optimization experiment, however, does not tell us how the profit is distributed over the 20 years; to find that out, you can copy the best parameter values into a simulation experiment we created before and run it to see the charts).

The best parameter values (3 and 4) are in the middle of the searched ranges, which means that the ranges were, most likely, well chosen.

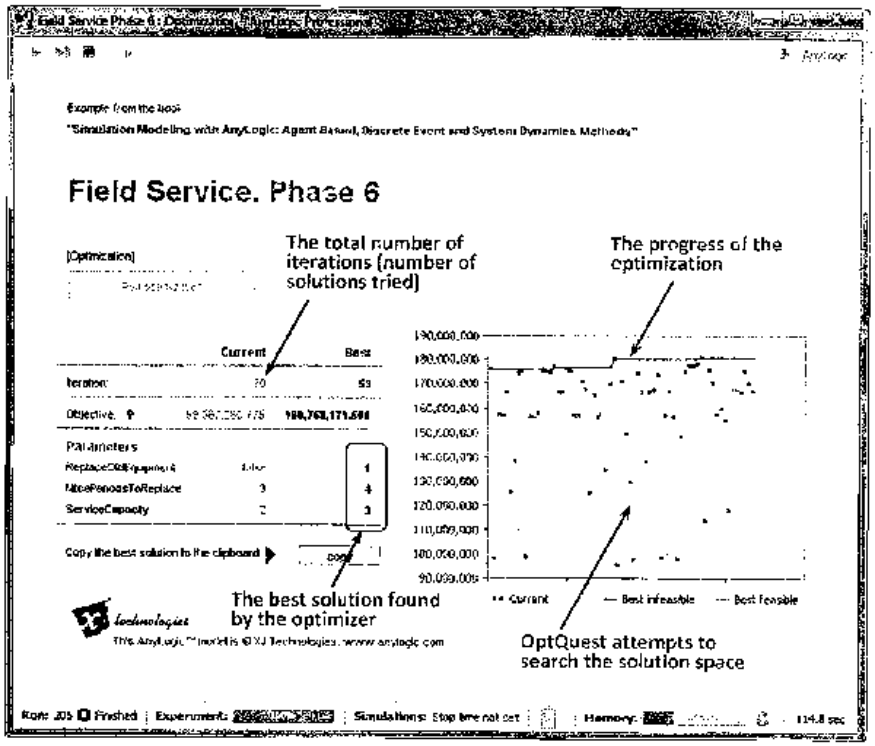


Figure 4.39 The statistics and the objective function needed for optimization

You may have noticed that, although the upper limit of the number of iterations was set to 2000, the optimizer only performed 70. This is because in our model, all three parameters are discrete and the solution space has only [2..8] * [2..6] * |true, false| = 70 points. The optimizer was able to *fully search the solution space* – a rare simple case.

We can, however, further reduce the number of iterations. Varying **MtcePeriodsToReplace** does not make any sense when the **ReplaceOldEquipment** policy is inactive. To pass this knowledge on to the optimizer you can use *constraints* – conditions on the input parameters that are tested *before a simulation run* and prevent the optimizer from trying particular solutions.

In AnyLogic, constraints are specified on the **Constraints** page of the optimization experiment properties in the form *<arithmetic expression over parameters> <relation> <bound>* where *<relation>* can be >=, <= or =. Boolean parameters in the constraint expressions take values 0 and 1.

Restricted solution space

MtcePeriodsToReplace	ReplaceOldEquipment = 0	ReplaceOldEquipment = 1
2	yes	yes
3	no	yes
4	no	yes
5	no	yes
6	no	yes
7	no	yes
8	no	yes

Value of the constraint expression

MtcePeriodsToReplace	ReplaceOldEquipment = 0	ReplaceOldEquipment = 1
2	2	-8
3	3	-7
4	4	-6
5	5	-5
6	6	-4
7	7	-3
8	8	-2

Constraint in AnyLogic

Constraints on simulation parameters (are tested before a simulation run):

Enabled	Expression	Type	Bound
☑	MtcePeriodsToReplace - 10 * ReplaceOldEquipment	<=	2.0

Figure 4.40 Optimization constraint

In our case, we need to invent an arithmetic expression that would be greater (or less) than a certain value for all meaningful combinations of the two parameters, see Figure 4.40 (when the replacement policy is inactive we just allow one arbitrary value of **MtcePeriodsToReplace**, say, 2). A possible expression that does the job is shown in the screenshot.

With that constraint enabled, the number of iterations reduces to 40 and the solution is found in 62.8 sec instead of 113.4 sec (on the author's machine with two cores).

4.8. Assumptions

We can now deliver the final results of the modeling project to our client. The deliverables, however, should always include the list of assumptions made by the modeler under which the results make sense.

In a properly performed project, each of those assumptions should have been discussed with the client and approved at the model design phases. Once again, we recommend involving the client in the project on all stages and making as many iterations with the client and getting as much feedback from the client as possible.

So, our major assumptions in this model are:

1. The formula for the equipment failure rate is one of our central assumptions, along with the exponentially distributed time between failures.
2. Repair, replacement, and maintenance times are triangularly distributed.
3. Repair, replacement, and maintenance have a fixed flat cost.
4. There are no roads in the area. Service crews drive in straight lines directly from origin to destination at a constant speed. The more correct formulation of this assumption is "the driving time is always proportional to the straight line distance from origin to destination".
5. There are no shifts or breaks. All service crews work 24 hours a day.
6. Service crews never fail and always have all necessary parts and tools on board. They never need to drive to the base location to pick up missing stuff.
7. Service crews do not optimize their routes. They just take the next request from the queue and drive there, whereas they could choose a request from a closest location.
8. Service crews are equipped with a radio and can take new assignments while driving.

4.9. Bonus phase. 3D animation

As a bonus to the work we have completed, we will build a 3D animation of the field service model. All things that happen in the model happen in 2D space. The goals of adding 3D are:

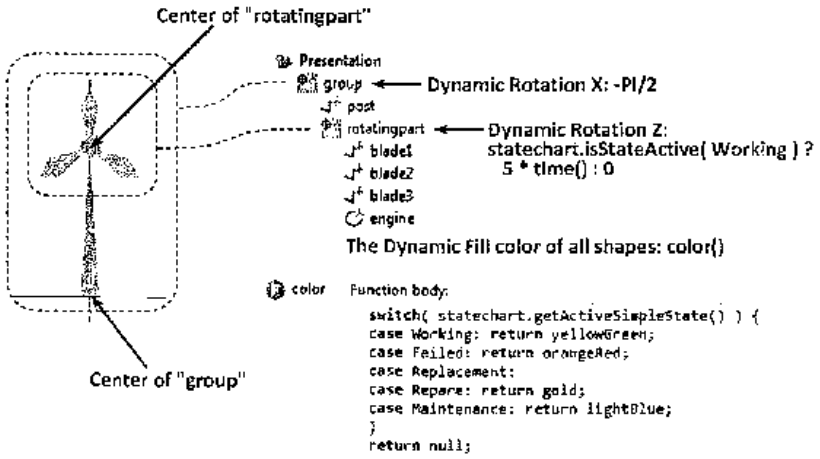
- To have fun.
- To make the model demonstration more entertaining and, sometimes, more convincing.
- To have a better debugging tool.

3D animation has one interesting advantage over 2D animation. With its perspective view, 3D animation is better suited for observing large areas. You can position the camera to have a detailed view of the current point of interest, and still be able to see the rest of the scene; the further the object is from the point of interest, the smaller it appears.

Adding the third dimension is easy. First, the group containing the equipment and service crew animations should be marked as shown in 3D (the **Show in 3D** property should be selected). We will use the framing rectangle as a ground, so it will get some fill color; its Z coordinate will be set to -1, and the Z-height – to 1. The home rectangle will be placed slightly above: Z = 1 and Z-height = 1. We will also create a

3D window and yet another view area for it. In total, we now have three views of the model: 2D, output, and 3D.

The animation of the equipment unit should also be marked as 3D. If you have used shapes not supported in 3D, you need to replace them with other shapes. For example, in our wind turbine picture, curves should be replaced by polylines. Also, we will rotate the picture by 90 degrees around the X axis to make it stand vertically in 3D (see also Example 14.2: "Rotation in 3D – a sign on two posts"). To have more fun, we can make the blades rotate when the wind turbine is working. Lastly, we can use color to reflect the state of the equipment.



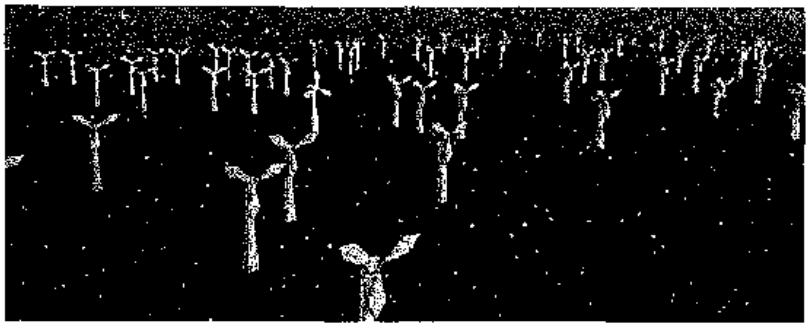


Figure 4.41 3D animation of the wind turbines

The shapes, groups and uses for the wind turbine animation and their properties are shown in Figure 4.41. The Z coordinates and Z-heights of the shapes were adjusted. A new function color() was created in the EquipmentUnit agent. It returns the

color reflecting the current state of the equipment, and is used in multiple dynamic **Fill color** fields of shapes.

If you run the model with these changes, you will notice that equipment animation is shown only in 3D windows; it has disappeared from the 2D animation. This is because the shapes are rotated 90 degrees around the X axis. To restore the 2D picture, we recommend you:

- Make a copy of the whole top-level group.
- Uncheck its **Show in 3D** property.
- Clear its dynamic **Rotation X** field.

Frequently, it makes sense to create separate animations for 2D and 3D views.

In the **ServiceCrew** agent, we will delete the flat 2D animation of the lorry and add a 3D object **Lorry** from the **3D objects** palette. The scale of the object may need to be adjusted to fit the scene. The 2D view will show the 2D projection of the lorry, which looks fine.

That's it! One thing you can do to improve the default 3D view is to choose a good viewpoint, copy its settings to a camera, and associate the camera with the 3D window (see Section 14.6, "Cameras").

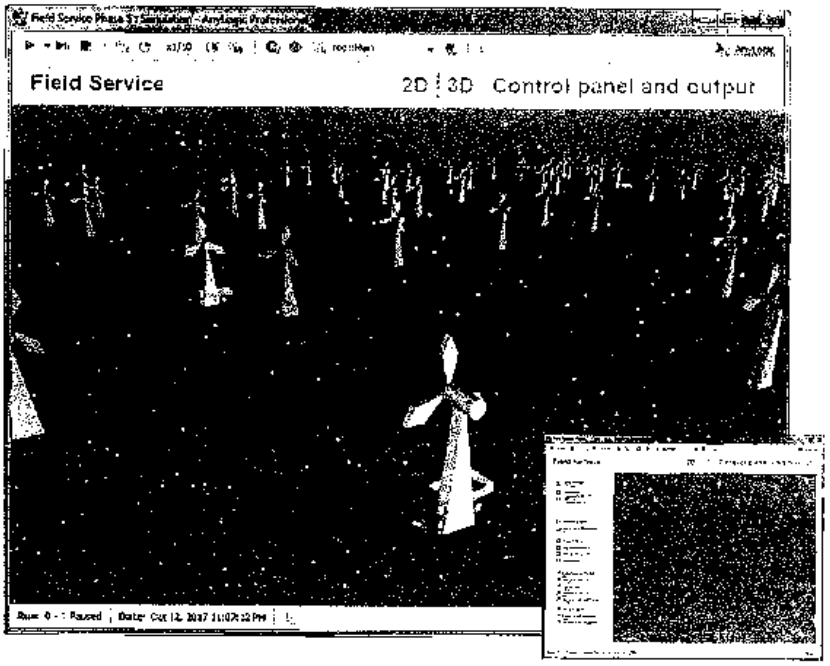


Figure 4.42 3D an 2D animation of the Field service model

4.10. Bonus discussion. Could we model this in discrete event style?

Now that we have finished the model, some of you may ask: Why did we use the agent based approach? Couldn't we do the same model in the discrete event style with much less effort? Your doubts may be partially supported by the fact that the system we were modeling is a service system, a type which is known to map well to process-based flowchart languages. Let us consider a potential discrete-event model design for the same problem.

The top-level process flow is easy. We have an incoming stream of maintenance and repair requests (which are entities) that are serviced by a pool of service crews (which are resources). This is a classical source – queue – service – sink model, see Figure 4.43 A.

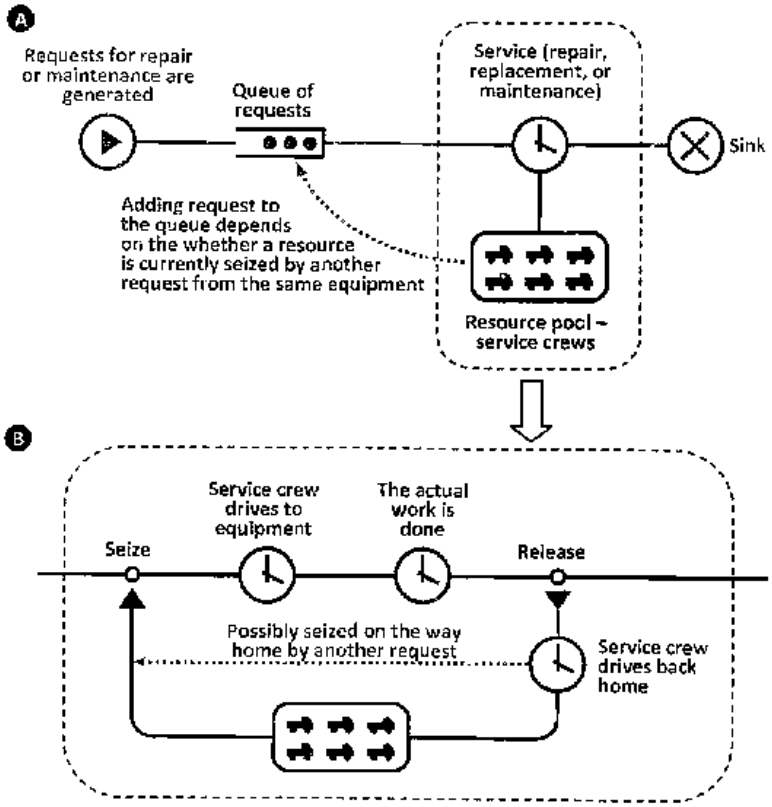

A
Requests for repair or maintenance are generated
Queue of requests
Service (repair, replacement, or maintenance)
Sink
Adding request to the queue depends on the whether a resource is currently seized by another request from the same equipment
Resource pool – service crews
B
Service crew drives to equipment
The actual work is done
Seize
Release
Possibly seized on the way home by another request
Service crew drives back home

Figure 4.43 The top-level process flow and the "service" subprocess

We can have a single queue for both types of requests and treat repair requests as high priority (all discrete event tools support priority queues). The rule that "an incoming repair request discards a maintenance request from the same unit" can also be implemented, provided you have access to the queue elements and the ability to remove them. However, another rule that says "if a request comes in while a service crew is already handling the equipment, the request is ignored" could require some non-trivial coding in a discrete event model; we would have to either search all entities being serviced or iterate through all service crews to find out what are they doing.

In a discrete event model, implementation of the queuing policy would result in approximately as much code as in our agent based model. Some DE tools may not have the necessary API. However, priority queue is a standard block in DE tools.

The "service" sub-process includes seizing a resource (a service crew), waiting for the crew to arrive, the actual work, and releasing of the service crew, which then drives home, see Figure 4.43 B. There are DE tools that support mobile resources. The only thing that is rarely supported is "interception" of the resource on its way home to assign another task (the red arrow in Figure 4.43 B).

So far there are no *conceptual* problems with using the discrete event method, only potential technical limitations of some particular tools. But, we still have to consider the "work" block and the request generation.

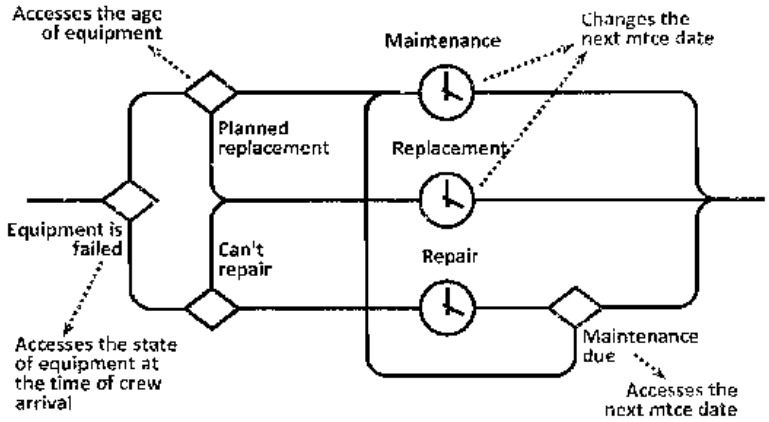


Figure 4.44 Subprocess "The actual work"

Once the crew arrives at the equipment unit, it may find the unit in different states. (Regardless of the request type: for example, while the unit was waiting for maintenance, it could fail.) The work being done also depends on the age of the equipment and on the date of the next scheduled maintenance. Conditions in the flowchart decision blocks explicitly refer to the *state of the equipment*, and the actions modify the state, see Figure 4.44.

The *state information*, therefore, should be stored for each equipment unit in a discrete event model anyway.

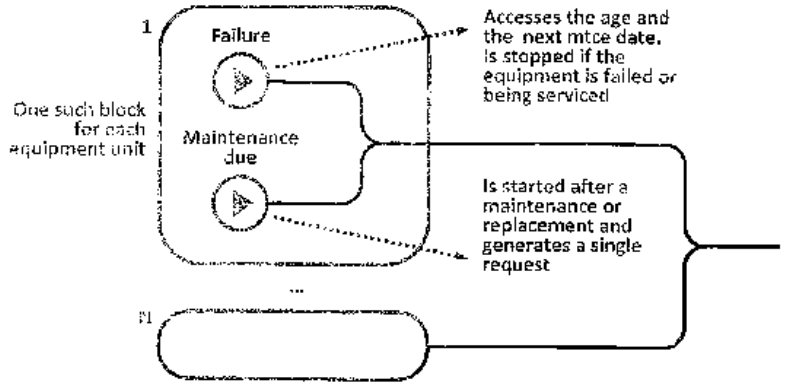

1
Failure
Maintenance due
One such block for each equipment unit
Accesses the age and the next mtce date. Is stopped if the equipment is failed or being serviced
Is started after a maintenance or replacement and generates a single request
...
N

Figure 4.45 Generator of repair and maintenance requests

The request generation part is the most interesting part. It does not map to the discrete event methodology at all. First, we clearly need a separate generator for each equipment unit because the failure rate and the maintenance cycle are defined for a unit. Second, the streams of requests are irregular: the next failure is scheduled only after the equipment has been brought back to the working state, and the next maintenance is scheduled only after the current one is completed, or if the equipment has been replaced. The request sources work, therefore, in a very unnatural mode: they are partially controlled by the changes of the equipment state that are done elsewhere in the flowchart. *The flowchart gets cross-linked by code references and stops being a self-explanatory visual construct.*

No matter how hard we try to "package" this system into a process, we will inevitably end up with mimicking a set of independent objects having state information and state-dependent behavior. And for that, the discrete event modeling paradigm does not offer anything nearly as elegant as agents and statecharts.

Chapter 5. System dynamics and dynamic systems

The intention of this chapter is not to teach system dynamics modeling (there are other excellent books serving this purpose), but rather to explain how to build and run system dynamics models in AnyLogic. At the end of the chapter, we will also briefly consider modeling of physical dynamic systems.

The process of building a system dynamics model in AnyLogic does not differ much from the process used in Vensim™, Powersim™, or STELLA™. AnyLogic supports:

- Stock and flow diagrams with automatic consistency checking
- Arrays (subscripts) with enumeration- and range-type dimensions
- Table functions (lookup tables)
- Delays and other SD-specific functions
- Units and unit checking

AnyLogic simulation engine includes numerical solver for differential, algebraic, and mixed equations.

The experiment capabilities include general simulation, interactive simulation (flight simulators and games), compare runs, sensitivity analysis, calibration and optimization, and Monte-Carlo.

In addition to this traditional toolset, AnyLogic offers several important extensions:

- You can build modular, hierarchical, and object-oriented system dynamics models. System dynamics components can be packaged into AnyLogic active objects, parameterized, organized in various structures, and reused.
- System dynamics can be combined with discrete event and agent based modeling (see Chapter 6, "Multi-method modeling"). AnyLogic naturally supports the interaction of stock and flow structures with events, statecharts, process flowcharts, and agent populations.
- AnyLogic simulation engine is a hybrid engine designed for efficient and accurate simulation of continuous dynamics being interrupted by a large number of discrete events.
- Animation capabilities are much richer than those in any other SD modeling framework and include 2D and 3D animation, interactive UI, and business graphics.
- AnyLogic models are standalone, 100% Java applications, and therefore run on any platform; they can even be published on the web as applets. This means you can easily deliver or share models with end users, who will be able to run them without installing any software.

5.1. How to draw stock and flow diagrams

Elements of stock and flow diagramming can be found in the **System Dynamics** palette, see Figure 5.1. If the user stays within the pure system dynamics modeling paradigm, these elements, perhaps along with the charts from the **Analysis** palette and dimensions created in the **Projects** tree, will be all he needs.

Drawing stocks and flows

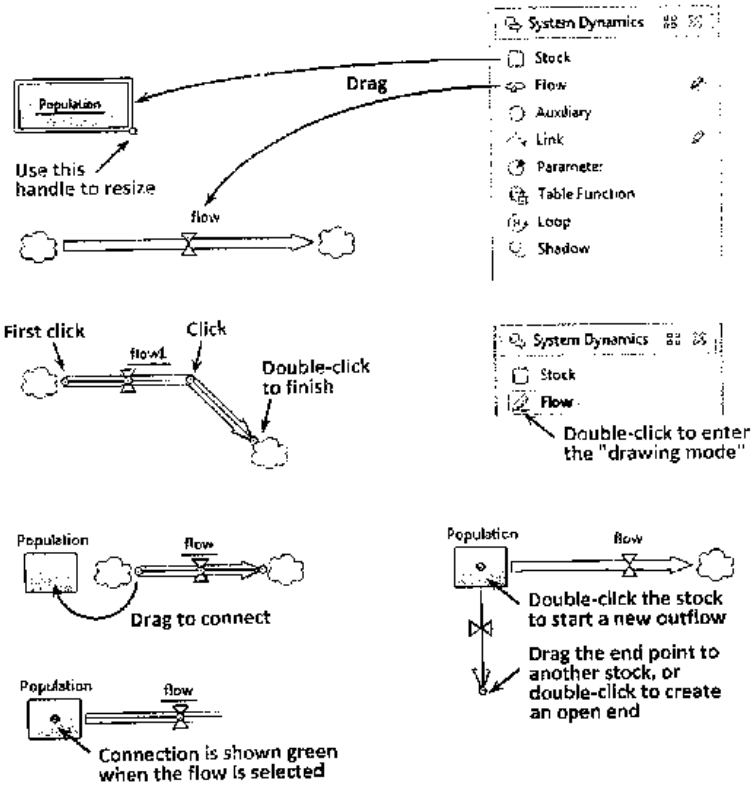


Figure 5.1 Drawing stocks and flows

▶ **To draw a stock:**

12. Drag the **Stock** object from the **System Dynamics** palette onto the canvas.
13. Enter the stock name (the name can be moved and changed at any time).
14. Resize the stock shape by dragging its bottom right corner.

▶ **To draw a flow from a stock:**

1. Double-click the stock and drag the flow out of it. Each subsequent click adds a point to the flow polyline.
2. To finish the flow, click on the target stock or double-click to create an open end (a "cloud").

▶ **To draw a flow not connected to any stocks:**

3. Drag the **Flow** object onto the canvas. A straight, open-ended flow "from cloud to cloud" is created.

▶ **To connect a flow to a stock:**

1. Drag its end point onto the stock. The connected end point turns green.

▶ **To draw a flow with multiple segments (a polyline flow):**

1. Double-click the flow icon in the **System Dynamics** palette. The flow tool switches to "drawing mode".
2. Draw the flow by clicking at each point. Double-click at the end point to finish.

▶ **To add a new point (segment) to an existing flow:**

1. Double-click the flow where you want to create a new point.

The color of stocks and flows can be customized by using the **Color** control in the stocks and flows' properties.

Drawing variables, dependency links, polarities, and loop types

In addition to dynamic variables that are not stocks or flows (also called auxiliary variables), you can consider using parameters and "constant" variables, which serve as inputs to the feedback loop structures.

▶ **To create a variable:**

1. Drag the **Dynamic Variable** object from the **System Dynamics** palette onto the canvas.
2. Enter the variable name (the name can be moved and changed at any time).

▶ **To create a dependency link from a variable**

1. Double-click the variable and drag the link.
2. Click to finish.

Links from stocks, flows, and parameters cannot be drawn by double-clicking these elements. You must create a link first, and then connect it.

▶ **To create a "standalone" link:**

1. Drag the **Link** object from the **System Dynamics** palette onto the canvas.

To connect a link to a stock, flow, parameter, or auxiliary variable:

1. Drag the end point of the link to the object. The connection is shown as a green point.

To create a loop icon:

1. Drag the **Loop** object from the **System Dynamics** palette onto the canvas.
2. Choose the direction and type of loop in the loop properties.

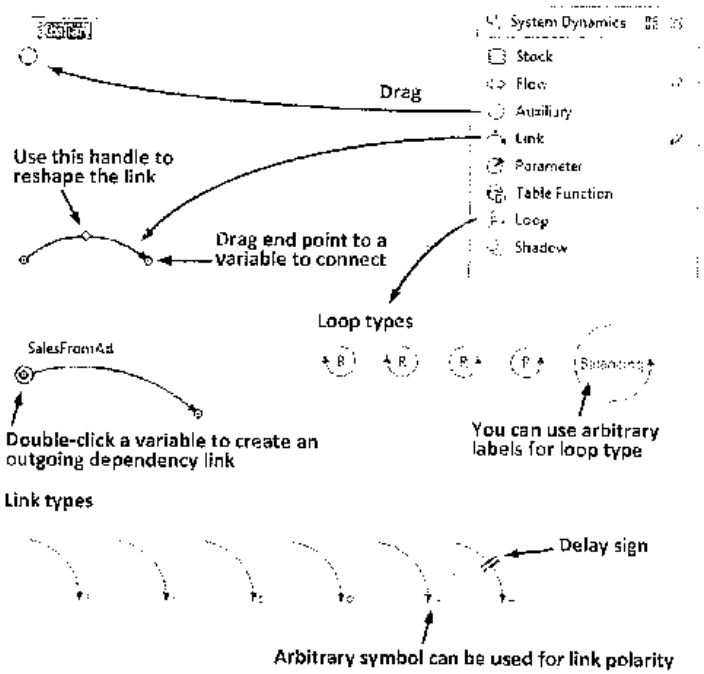


Figure 5.2 Drawing stocks and flows

AnyLogic offers a choice of link polarity symbols, see Figure 5.2. A link can also be decorated with the delay sign. Loop signs can be customized as well. The colors of variables, links, and loops, and link line width can be changed in these elements' properties.

Unlike in some other graphical notations, in AnyLogic table functions are not graphically linked to variables.

Naming conventions for system dynamics variables

System dynamics variables should obey AnyLogic naming conventions (originating from Java, see Section 10.7). Those are a bit different than in other system dynamics tools.

Variable names cannot contain spaces or line breaks. We recommend using mixed case, with the first letter of each word capitalized. Underscores ("_") are also allowed.

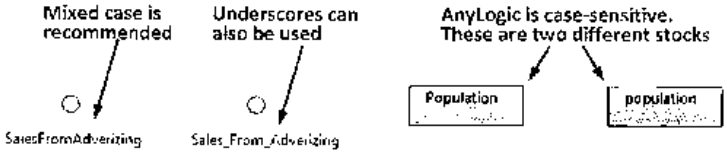


Figure 5.3 Names of system dynamics variables

Layout of large models. "Sectors" and shadow variables

Traditional SD frameworks offer the following way of laying out large models: the model is partitioned into pieces called *sectors*, each focusing on a particular aspect - for example, Housing, Business, Tax, and Labor. The diagram for each sector is drawn separately from the others, and variables used in multiple diagrams are multiplied as well, so that there are *no graphical links between sectors*. For each such variable, there is one "original" instance in one of the sectors and "shadow" copies in other sectors.

In AnyLogic, you can partition the model into components in an object-oriented way by using active objects exposing the "input" and "output" dynamic variables as part of their interface. However, the traditional "sector" method is still available. For that, AnyLogic supports view areas (see Example 13.10: "Hyper link menu to navigate between view areas") and *shadow variables*. In Figure 5.4, the population model is partitioned into a Housing sector and a Population sector. The "interface" between the sectors includes two variables:

- The stock **Houses** from the Housing sector is used in the Population sector.
- The variable **HouseholdsToHousesRatio** from the Population sector is used in the Housing sector.

Correspondingly, the Housing sector has a shadow copy of **HouseholdsToHousesRatio**, and vice versa. You can distinguish the shadow from the original by the angle brackets around its name: **<HouseholdsToHousesRatio>**.

▶ **To create a shadow variable:**

1. Drag the **Shadow** object from the **System Dynamics** palette on the canvas.
2. Select the "original" variable from the list.

You can read more about view areas in Chapter 12, "Presentation and animation: working with shapes, groups, colors".

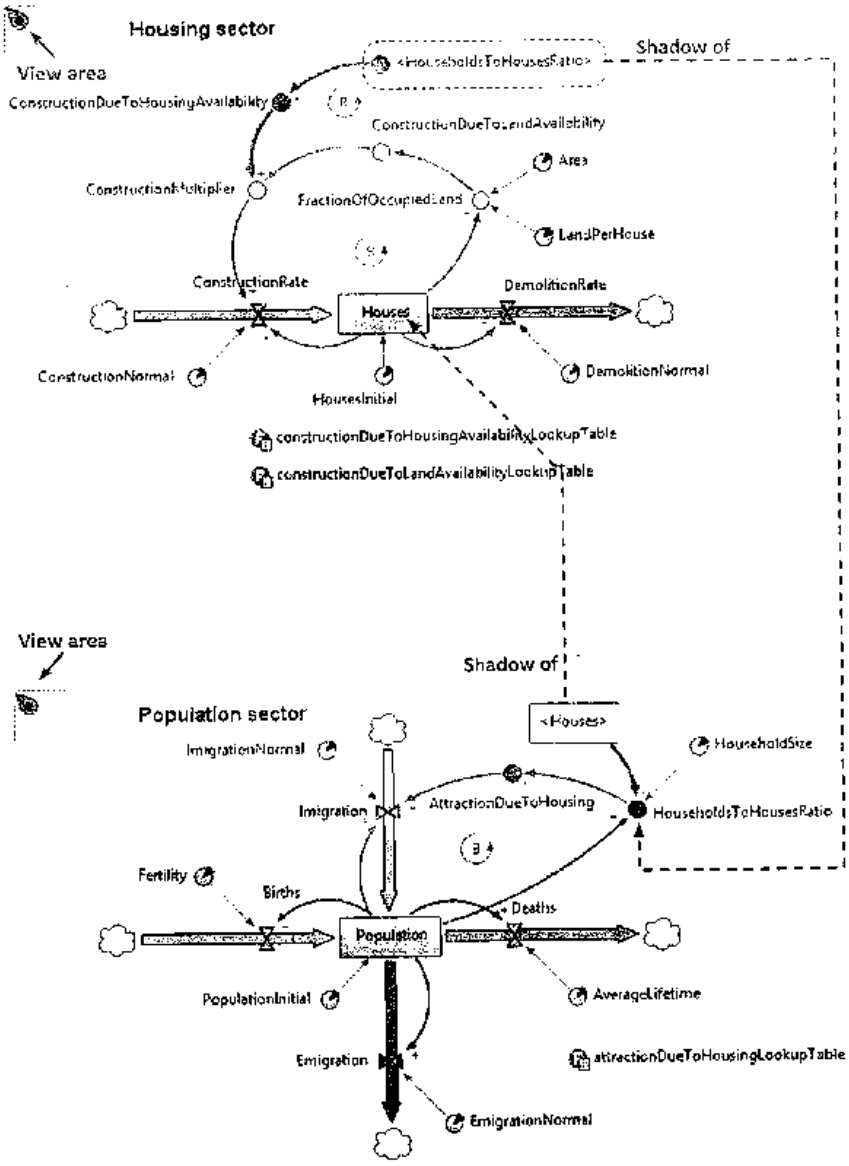


Figure 5.4 Shadow variables and view areas are used to create "sectors"

5.2. Equations

In AnyLogic, equations for stocks are constructed automatically from the graphical structure, and optionally can be typed manually. Equations for flows and other variables are entered in their properties and checked against the existing dependency links.

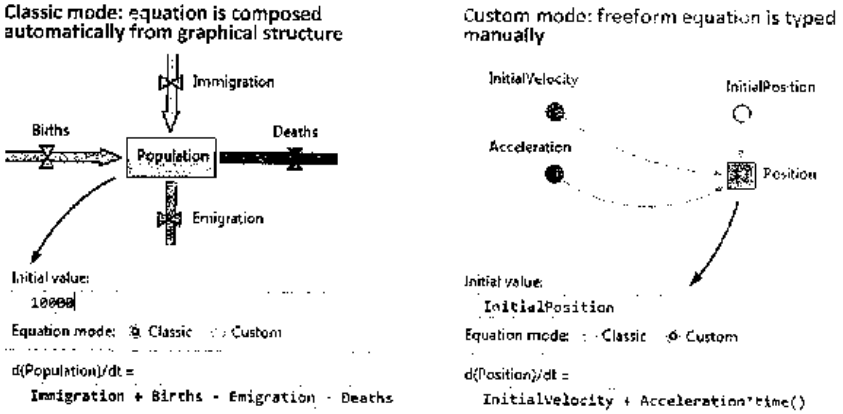


Figure 5.5 Classic and custom modes of stock equations

The default mode for stock equation is classic (automatic). In this mode, each incoming flow is added to the stock derivative expression, and each outgoing flow is subtracted. Therefore, the resulting equation is a linear combination of flows always conforming with the graphical structure, see Figure 5.5.

If you switch the radio button **Equation mode** in the stock properties to **Custom** mode, you will be able to manually type a freeform expression. The expression can then contain arbitrary arithmetic operations and function calls, see Figure 5.5 on the right. This mode is used more frequently in dynamic (physical) systems models than in system dynamics.

The expression for stock derivative typed in custom mode is still checked for conformance with the graphics – namely, for each instance of a variable in the expression there should be an incoming link or flow from that variable to the stock, and vice versa.

Equations for flows and other variables are freeform. They are typed manually and checked for conformance with the graphical structure in the same manner, see Figure 5.6. If discrepancy is detected, a problem item appears in the **Problems** view and a small red sign is displayed next to the equation. To view the error description, move the mouse cursor over the sign.

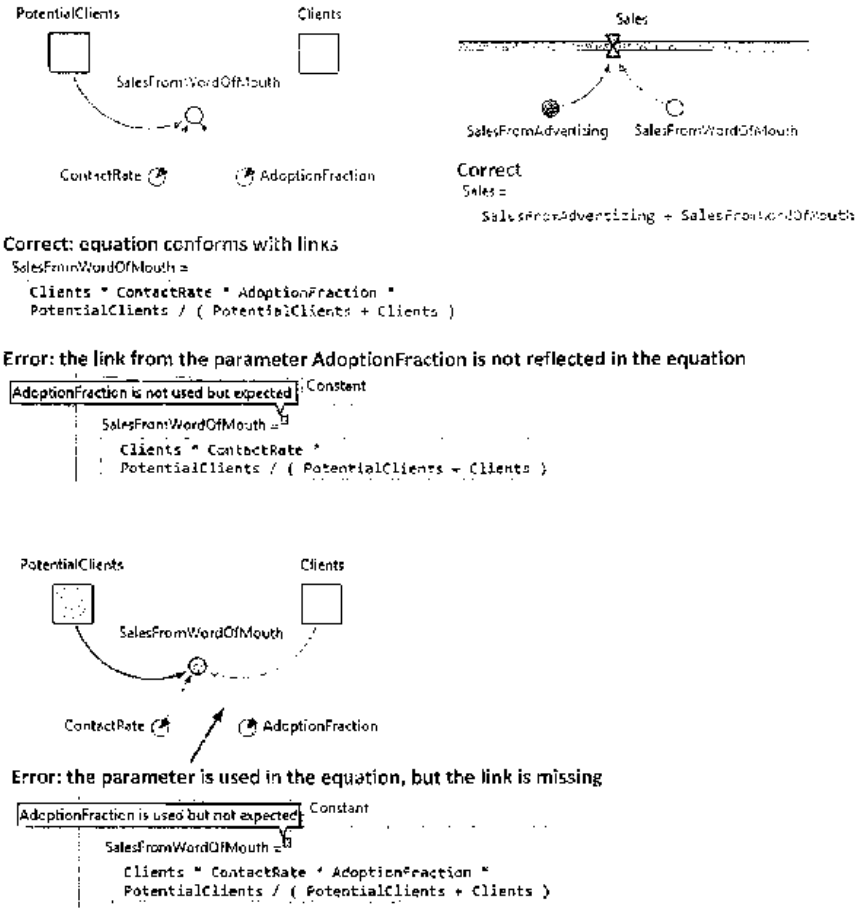


Figure 5.6 Equations of flows and other variables

When typing equations, it makes a lot of sense to use AnyLogic code completion (see Section 10.4). You do not have to type the name of a variable completely (which may be time-consuming, as the names in system dynamics tend to be long). Instead, you can type the first couple letters of the name and press Ctrl+space. AnyLogic will display a list of suggestions – variables and functions whose names start with the typed letters. You then just need to choose the intended item, see Figure 5.7.

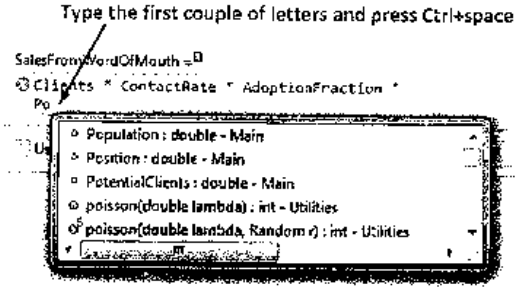

Type the first couple of letters and press Ctrl+space to see the choices
SalesFromWordOfMouth =
Clients * ContactRate * AdoptionFraction *
Population : double - Main
Position : double - Main
PotentialClients : double - Main
poisson(double lambda) : int - Utilities
poisson(double lambda, Random r) : int - Utilities

Figure 5.7 Code completion

▶ **To rename a system dynamics variable:**

1. Select the variable.
2. Double-click the variable name in the graphical editor and enter the new name, or type the new name in the variable properties.
3. Press Ctrl+Enter to perform the model refactoring.

After renaming a system dynamics variable, you should press Ctrl+Enter before leaving the name editor. This will rename all occurrences of the variable in all expressions throughout the model. If you do not press Ctrl+Enter, the variable name will change, but its occurrences will not be renamed.

Using Java in SD equations

Manually typed equations are in fact Java expressions (see Section 10.5) of numeric type double. As such, they can contain function calls and conditional operators (see Section 10.5), as well as reference arbitrary objects in the model. This is an example of the conditional operator:

```
Level < 10 ? MaximumRate : NormalRate
```

The above expression evaluates to **MaximumRate** when the **Level** is lower than 10 and to **NormalRate** otherwise. This is equivalent to the Vensim™ IF THEN ELSE function.

You can define custom Java functions (see Section 10.4) implementing analytical or general algorithmic dependencies and use them in SD equations. The functions should return the Java type **double**. For example, suppose a dependency of a certain form is used in multiple places across the model. You can create a Java function with one or several arguments – say, **SmoothReverseProportional()** – and use it in multiple equations:

```
MaxBirthRate * SmoothReverseProportional( Crowding )
```

The following expression references the collection of consumers (agents) located at the same level as the system dynamics diagram, and uses the statistics function **NoWaiting()** defined in that collection:

```
BaseRate + consumers.NoWaiting() * DemandCoefficient
```

Assume the system dynamics diagram is located in an active object, which is embedded in **Main**, and that there is another active object, **manufacturing**, embedded in **Main** as well, with a queue named **stock** inside. To calculate, for example, the total cost of entities in **stock**, you should write in the system dynamics model:

```
get_Main().manufacturing.stock.size() * CostPerItem
```

As you can see, the use of Java in equations helps you to link the system dynamics with other objects in the model, in particular with agents and discrete event objects. For more information on how to navigate in the hierarchical model with Java, see Section 10.9, "Where am I and how do I get to...?".

Keep in mind, however, that linkage of *two system dynamics models located in different active objects* should not be done with Java expressions. Instead, you should explicitly link the models using external variables and graphical connectors.

You should also refrain from using stochastic functions – functions that return a different random value each time they are called, like **exponential()** or **logistic()** – in the system dynamics equations.

"Constant variables" and parameters

Any dynamic variable (except for a stock) can be declared *constant* by selecting its **Constant** property. The expression typed in its value field will be treated as an initial value, and will not be evaluated on each numeric integration step. You can, however, change the value of such a variable by explicitly assigning a new value.

Parameters, in this sense, are similar to constant variables. They have no dynamic equations and have default values that can be changed during the model runtime. In addition, parameters are a part of the active object interface (they are visible from outside) and can be linked to the parameters of outer and inner active objects.

In Figure 5.8, the variable **Acceleration** is marked as constant. Its initial value is 1. At time 10 the event (see Section 8.2) **ChangeAcceleration** will occur and change the value of **Acceleration** to 2. This, by the way, is yet another way of linking a discrete element of the model with the system dynamics.

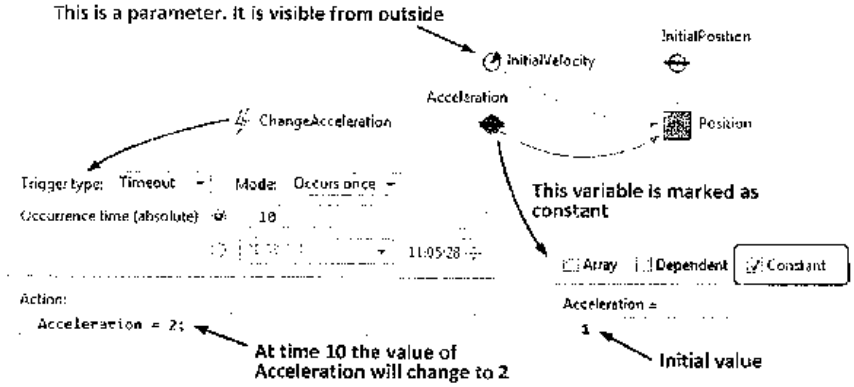


Figure 5.8 Constant variable and event changing its value

Units and unit checking

Each value in the model is measured in certain *units* (or is unitless). You can define new base units; specify units for variables, parameters, and functions; and check if the units are consistent with the formulas. Although these steps are optional and do not affect the simulation results, they are important for keeping track of the physical meaning of the variables and for model verification purposes. Units may be obvious for some variables and not so obvious for others. The next section includes an example of unit specification.

▶ **To specify units for a variable, parameter, or function:**

1. Select the checkbox **Use units** in the variable/function properties.
2. Type the expression for the unit.

The expression can contain the predefined unit **time**, any unit that has previously been defined in the model, or a new unit. The expression may also include the * and / operations. If you leave the expression field blank, the variable will be considered as unitless. Some typical examples of units are shown in Figure 5.9.

▶ **To perform unit checking throughout the model:**

1. Select **Tools | Check model units** from the main menu. The results are displayed in the **Problems** view.

The unit inconsistencies are shown as errors. The instances of a variable with undefined units (one with the checkbox **Use units** not selected) in an expression of a variable with defined units is signaled as a warning, see Figure 5.9.

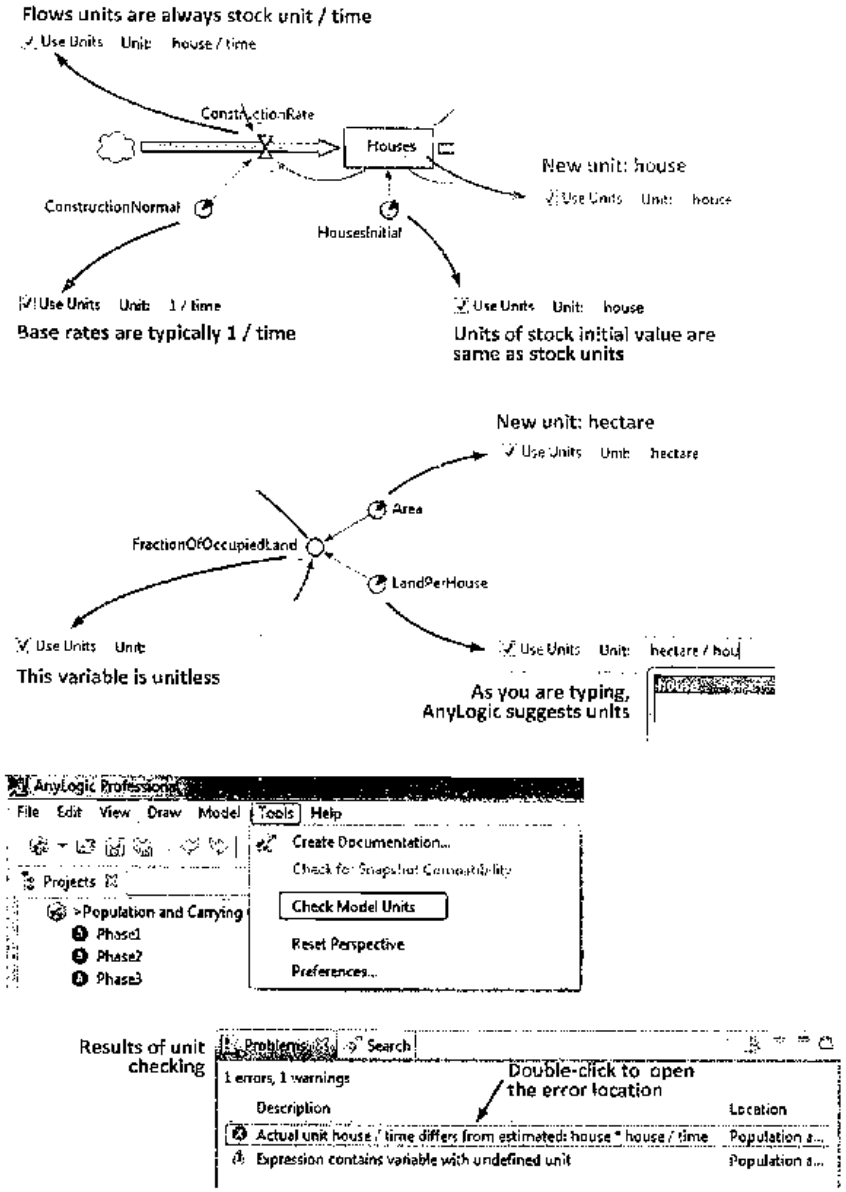


Figure 5.9 Units and unit checking

5.3. Example: Population and carrying capacity

Now, we will build a simple model of population dynamics in a habitat with limited carrying capacity.

"The *carrying capacity* of any habitat is the number of organisms ... it can support and is determined by the resources available in the environment and the resource requirements of the population." (Sterman, 2000)

In our simple model, we will assume the carrying capacity is constant.

Phase 1: Unlimited resources. Positive feedback. Exponential growth

There will be just one stock in the model – the **Population** stock. We will assume a closed system without immigration and emigration, so **Births** is the only inflow to the stock, and **Deaths** is the only outflow. In the first version of the model, we will assume environmental resources are unlimited. Therefore, the birth rate (the average number of new individuals produced by an individual per year) is constant and is at its maximum level. So is the average lifetime.

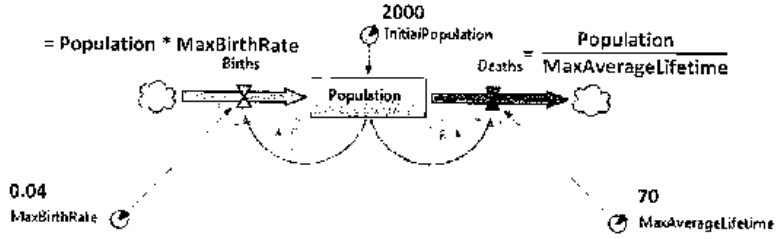


Figure 5.10 The population model. Unlimited resources

▶ **Create the stock and flow diagram:**

1. Create a new model.
2. Open the **System Dynamics** palette and drag the **Stock** onto the canvas. Change its name to **Population** and resize the stock so that the name can be placed inside.
3. Double-click the stock and drag out the flow to the right. Double-click to finish. Name the outflow **Deaths**.
4. Drag the **Flow** from the **System Dynamics** palette and place it so that its arrow links to the stock. This will be the **Births** inflow.
5. Drag the **Parameter** from the same palette and place it above the stock. Name the parameter **InitialPopulation**. Set the default value of the parameter to 2000.

6. Drag a **Link** so that its beginning connects to the parameter. Connect the end of the link to the **Population** stock.

The connections are indicated as green dots when the link is selected. When a variable is selected, the connected flows and links are highlighted in the color magenta.

7. Select the stock and type **initialPopulation** in the **Initial value** field of the stock properties.

You do not have to type all the letters of a name. Just start typing and press Ctrl+Space to open the code completion popup window (see Section 10.4). Then select the name you are looking for from the list.

8. Create two more parameters: **MaxBirthRate**, with the default value 0.04, and **MaxAverageLifetime**, with the default value 70. Place the parameters near the **Births** and **Deaths** flows correspondingly, as shown in Figure 5.10.
9. Open the properties of the **Births** flow and type the formula for births: **Population * MaxBirthRate** (remember to use code completion).

Notice the error message that appears once you finish typing the formula: "Population, MaxBirthRate is used but not expected". This message signals the inconsistency between the graphics and the formulas, namely the fact that the influence of the **Population** stock and the **MaxBirthRate** parameter on the **Births** rate is not "confirmed" graphically.

10. Create the two missing links in the same manner as in step 7. The error message disappears.
11. Similarly, enter the formula for the **Deaths** flow: **Population / MaxAverageLifetime**. Draw the corresponding links.

The first version of the model is ready to run. But first, let us discuss the meaning of the formulas. The formula for **Births** means that each individual produces another one, on average, every 1/0.04 = 25 years (notice that a couple produces twice as many), so each year there will be **Population * 0.04** births. The formula for **Deaths** means the following: an individual will "leak out" of the **Population** stock, on average, in 70 years, so its "individual outflow" is 1/70 per year. Given that there are multiple individuals, their total immediate outflow is **Population * (1/70) = Population / 70** deaths per year.

▶ **Run and explore the model:**

12. Run the model. You can see the current values of the variables displayed underneath their names. The values change as the model is simulated.
13. Click the **Births** flow to open its **Inspect** popup window. In the upper right corner of the **Inspect** window, click the chart icon to display the time chart of the variable.
14. Do the same for the **Population** stock and the **Deaths** flow. Watch the dynamics of the model until the time is approximately 100 years, then click the **Pause** button. The picture looks like that shown in Figure 5.11.

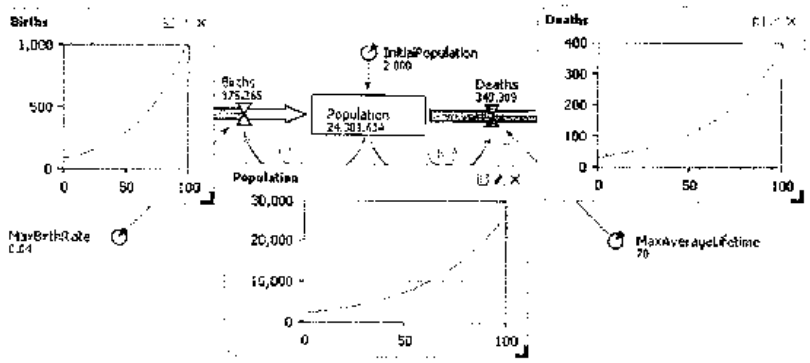


Figure 5.11 You can view the time plot of a dynamic variable in the Inspect window

While the model is executed in such a "slow" mode, you can play with the parameter values using the runtime editors in the **Inspect** window.

15. Open the **Inspect** window of the **MaxAverageLifetime** parameter and click the pen icon in its top right corner. Change the value to 12 years and continue the simulation. Watch the change in the system dynamics; this is explained later.

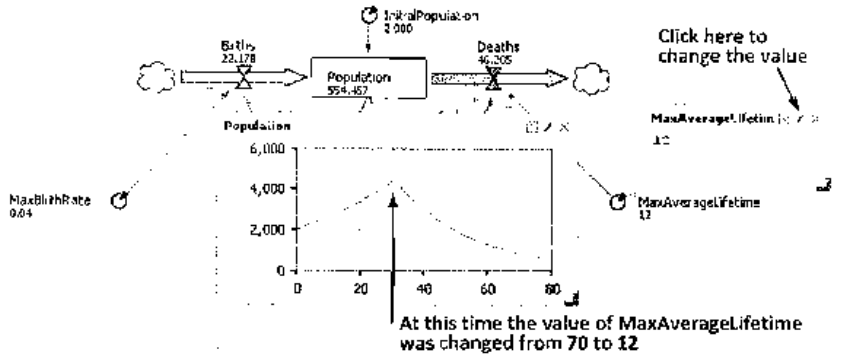


Figure 5.12 The balancing feedback dominates after the parameter change

By default, an AnyLogic model runs in a *scale to real time* (which, in our case, is one model year per second) and has *no specific stop time*. This is not typical for system dynamics modelers, who are used to instant calculations for a limited simulated time period. You can change both settings in the simulation experiment properties.

16. Select the **Simulation** experiment in the **Projects** tree and open the **Presentation** page of its properties. Set the **Execution mode** to **Virtual time**.
17. Open the **Model time** page and set the **Stop time** to 100.
18. Run the model again. This time, the simulation is performed almost instantly and the time charts are available right away.

The charts in the yellow popup windows are useful for rapid exploration of the model. However, you must re-open and re-adjust them in each model run. For the most interesting variables, you can add "permanent" charts.

▶ **Add the time plot of the key variables:**

19. Close the running model and go back to the editor.
20. Open the **Analysis** palette and drag the **Time plot** underneath the diagram.
21. In the properties of the time plot click **Add data item** and type **Population** in the **Value** field. Set the title of the item to Population, as well.
22. Open the **Appearance** page of the plot properties. In the **Legend** section, choose the position of the legend on the right of the chart.
23. Ctrl+drag the plot to create another one below, see Figure 5.13.
24. Delete the **Population** item from the second plot and add **Births** and **Deaths** items.
25. Add the third item to the second plot. Set the value of the item to **Births – Deaths** and the title to Net Birth Rate.
26. Run the model and view the plots.

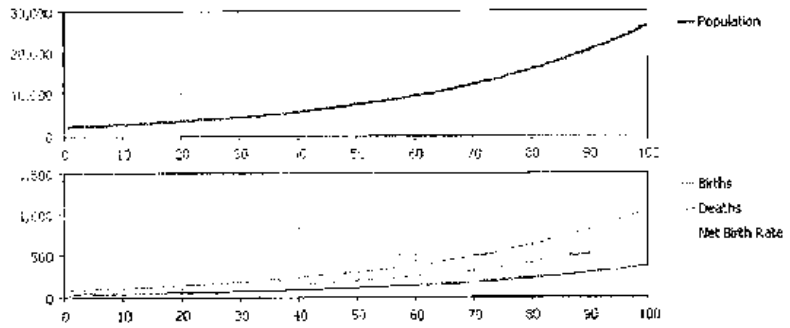


Figure 5.13 Time plots of the key system variables. Exponential growth

We observe *exponential growth* of the population. This exponential growth arises from *positive (self-reinforcing) feedback*. The larger the population, the more individuals are born, which further increases the size of the population. In our model, this positive feedback interacts with a *negative (balancing) feedback*: the larger the population, the more people die. Under the default parameter values, the net birth rate is positive, which means the reinforcing loop dominates and results in exponential growth. When you were experimenting with the value of the average lifetime, you saw the configuration where the balancing feedback dominated, see Figure 5.12.

A good rule for system dynamics modeling is to always indicate the link polarities and loop types.

▶ **Set link polarities and indicate the loop types:**

27. Select links one by one and choose their polarities.
28. Open the **System Dynamics** palette and drag the **Loop** icon inside the **Population -> Births -> Population** loop, as shown in Figure 5.10. Change the loop type to **R** (reinforcing).
29. Create another loop icon for the **Population -> Deaths -> Population** loop (this time the loop is balancing). Change the loop direction to **Counterclockwise**.

Customizing the dataset collection

In AnyLogic, you can customize the collection of datasets for dynamic variables that are displayed in the yellow **Inspect** windows and in the "permanent" charts. By default, there is a dataset associated with each dynamic variable. The dataset is updated at every time unit and contains the 100 latest samples. When you open the variable time plot in the **Inspect** window, this is the dataset that shows up.

▶ **To change the settings of the dataset auto-collection for dynamic variables:**

1. Open the properties of the active object where the variable is located (the settings are individual for each active object) and go to the **Advanced** page.
2. Choose whether you want to collect the datasets and, if yes, choose the recurrence time and the dataset length, see Figure 5.14.

The reason for not collecting the datasets is the impact on model performance and memory, which may only matter for very large models.

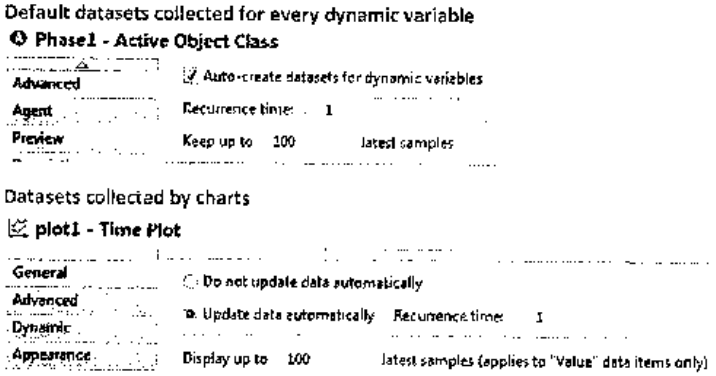


Figure 5.14 Customizing the dataset collection

The charts that are added to the model from the **Analysis** palette have their own settings, which can be changed at the bottom of the chart's **General** property page, see Figure 5.14.

Phase 2: Crowding affects lifetime. Negative feedback. S-shaped growth

Now we will introduce carrying capacity to our model. We will assume the environment can only support 5,000 individuals, and as the population gets closer to that number, lifetime reduces dramatically. When the population is significantly below the carrying capacity, lifetime is at its maximum level. How do we incorporate such dependency into the model? There are two ways:

- We can put together an analytic formula that has the desired shape.
- We can create a *table function* (*lookup table*) where we can build dependency of an arbitrary shape.

In this phase we will use a table function, and in the next phase we will use a formula to create a similar dependency for the birth rate.

The value, which will be the input to the table function, is the population relative to carrying capacity. We will create the corresponding variable and call it **Crowding**.

▶ **Update the feedback structure:**

1. Create a new parameter **CarryingCapacity** with the default value 5,000.
2. Create a new dynamic variable **Crowding**, which equals **Population / CarryingCapacity**.
3. Create the links from **CarryingCapacity** and from the **Population** stock to the **Crowding** variable (these links should be dragged from the palette). Set up the link polarities.
4. Create another variable **AverageLifetime**. This will be the current average lifetime, which changes dynamically as the crowding grows.
5. Drag the **Table function** from the **System Dynamic** palette and name it **EffectOfCrowdingOnLifetime**.
6. Enter the formula for **AverageLifetime: MaxAverageLifetime * EffectOfCrowdingOnLifetime(Crowding)**.
7. Change the formula of the **Deaths** flow to **Population / AverageLifetime**.
8. Correct the link from **MaxAverageLifetime**: it should now point to **AverageLifetime**.
9. Draw the missing links: double-click the **Crowding** variable and place the link arrow on **AverageLifetime**. Then, double-click **AverageLifetime** and create a link to **Deaths**. Set up the polarities.

In AnyLogic, links are drawn only between variables, but *not between functions and variables*. Therefore, although the function **EffectOfCrowdingOnLifetime** is used in the formula of **AverageLifetime**, the link between them is not drawn.

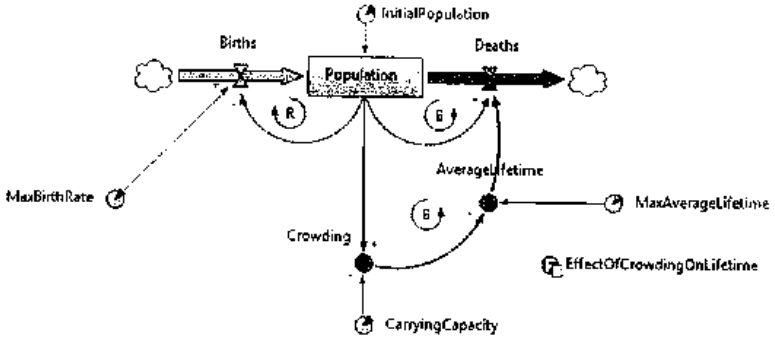


Figure 5.15 The feedback structure, updated to include the effect of crowding on lifetime

Now we need to fill in the argument/value table in the table function. On a real project, you could use historical data. In our exercise, we will make up the function

using common sense. Given the formula for **AverageLifetime**, the value range of the function is 0..1 because **AverageLifetime** cannot be negative or be greater than **MaxAverageLifetime**. We will assume that until the population reaches half of the carrying capacity, lifetime remains at its maximum level, at which point it goes down and reaches almost 0 when crowding is at the level of 1.5 (we could use 0 exactly, but then we should have protected the formula of **Deaths** from division by 0).

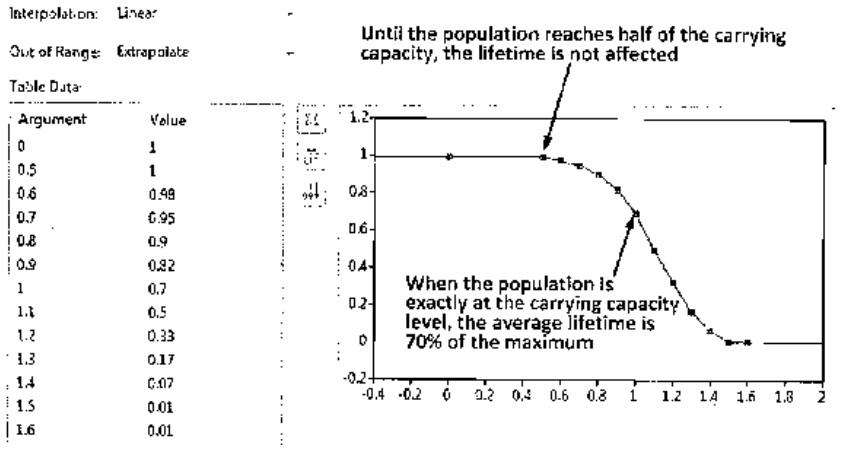


Figure 5.16 The table function EffectOfCrowdingOnLifetime

10. Open the properties of the table function and fill the table, as shown in Figure 5.16.
11. Set the **Out of range** option to **Extrapolate**. This is necessary because the argument of the function may potentially exceed the last value of 1.6, and we should tell the function how to handle it. The graph reflects the current inter- and extrapolation.
12. Run the model and see the new dynamics.

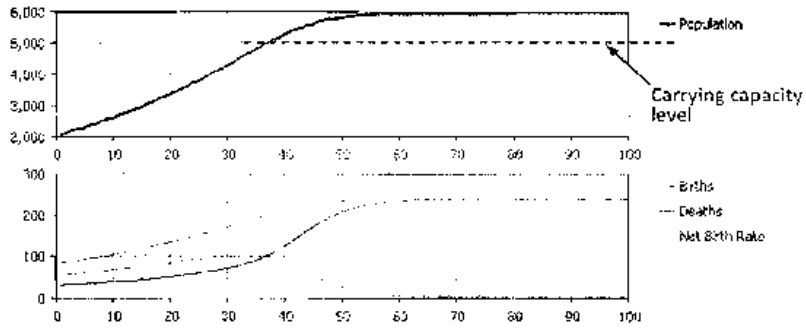


Figure 5.17 The S-shaped behavior of the updated population model

The time charts of the updated model are shown in Figure 5.17. The unlimited exponential growth has changed to *S-shaped growth*. This reflects the fact that the initially dominating positive feedback through births is gradually suppressed by the negative feedback through the effect of crowding on lifetime, which dominates as the system reaches equilibrium. The "power" of that second feedback is defined by the table function. Notice that equilibrium is reached not at the level of the carrying capacity, but above it. Our model, however, is not yet complete, for we have not yet defined the effect of crowding on births.

Phase 3: Crowding affects births

We will assume that crowding affects the birth rate in a similar way to how it affects lifetime: the more crowded the habitat, the fewer individuals born. This time, we will use an analytic function of shape similar to the table function in Figure 5.16. The function and its graph are shown in Figure 5.18. The formula structure has no specific relation to the population dynamics; it just has been chosen because of its graph shape.

AnyLogic supports exponential, logarithmic, power, and many other mathematical functions (see Section 10.4) due to its rich Java heritage.

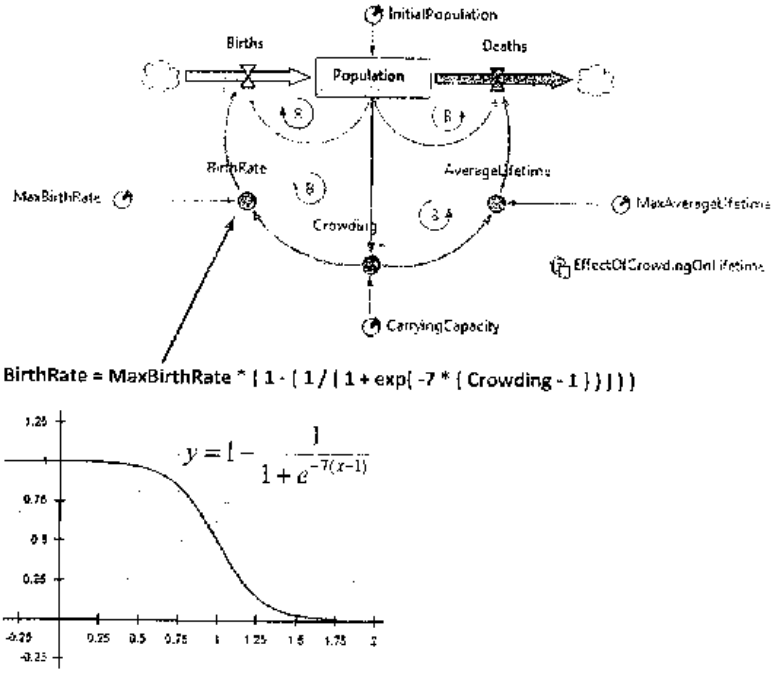


Figure 5.18 The effect of crowding on births is modeled by an analytic function

▶ **Update the feedback structure to include the effect of crowding on births:**

1. Create a new variable **BirthRate** and change the formula of **Births** to **Population * BirthRate**.
2. Type the formula for **BirthRate**, as shown in Figure 5.18.
3. Update the links structure according to the new dependencies.
4. Run the model.

The behavior is similar to the previous one (see Figure 5.18), but now the equilibrium is reached near the carrying capacity level: at about 4,980 individuals. Births and deaths stabilize at the level of ~100 per year, and average lifetime at about 50 years.

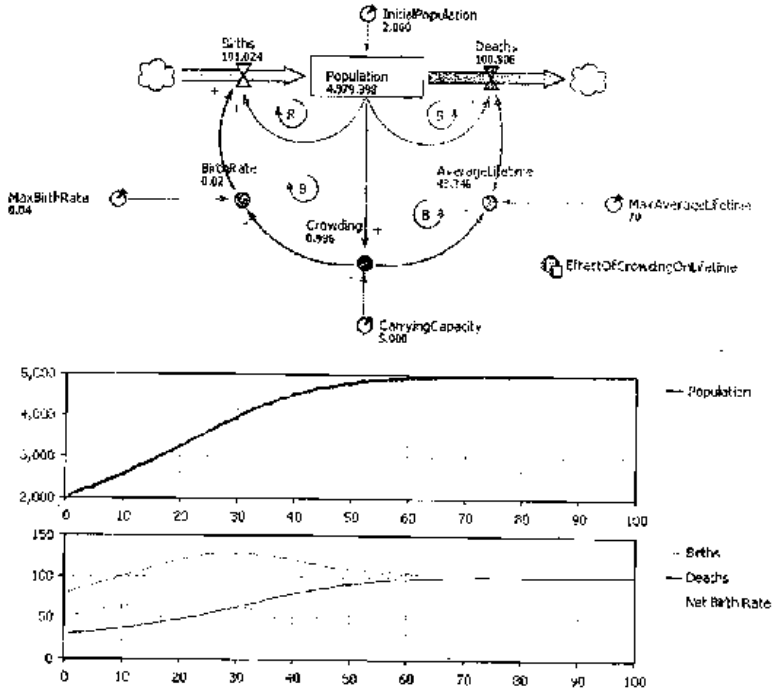


Figure 5.19 The S-shaped behavior of the population model with multiple negative feedbacks

You may wonder why the population size does not stabilize exactly at the level of carrying capacity. The answer is that the *parameter* **CarryingCapacity**, which has a value of 5,000, is not explicitly incorporated into the model as the goal, but affects the behavior via the two functions. The *actual carrying capacity* in our model is 4,980, as defined by the *combination of the parameter and the functions*. You can calibrate the shapes of the functions to achieve the exact value.

Phase 4: Negative feedback with delay. Overshoot and oscillation

Now assume that resource shortage does not affect births immediately, but instead after a certain time delay (this may happen due to some social or physiological reasons). We will incorporate the **delay()** function into the formula of the **BirthRate**.

▶ **Add delay to the effect of crowding on births:**

1. Add a new parameter, **MaturationDelay**, with the default value 15, and a link from this parameter to the **BirthRate**.
2. Modify the formula of the **BirthRate**, as shown in Figure 5.20.

3. Add the "delay decoration" to the link from **Crowding** to **BirthRate** (this is done in the link properties). This is an optional but recommended action.
4. Run the model.

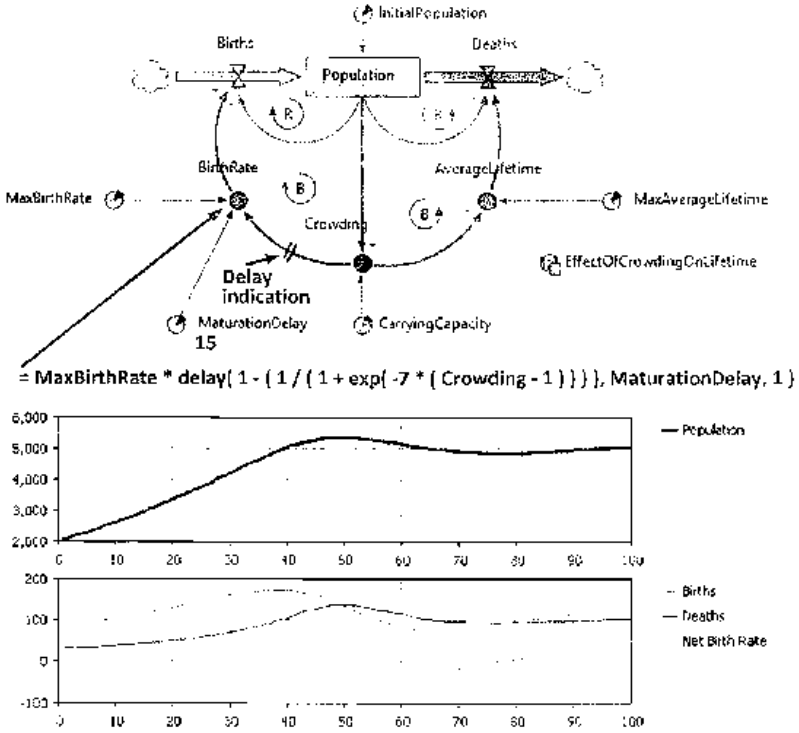


Figure 5.20 Delay in the negative feedback causes oscillation

"Time delays in the negative loops lead to the possibility that the state of the system will overshoot and oscillate around the carrying capacity" (Sterman, 2000), which we observe in our model (Figure 5.20).

The function **delay(x, T, x0)** reproduces the exact behavior of its argument **x** with the delay of **T** time units. The last argument, **x0**, is the value provided as output before the value of **x T** time units ago is known – that is, during the interval **0..T**, see Figure 5.21.

Besides this type of delay, AnyLogic supports several other types, including first and third order delays.

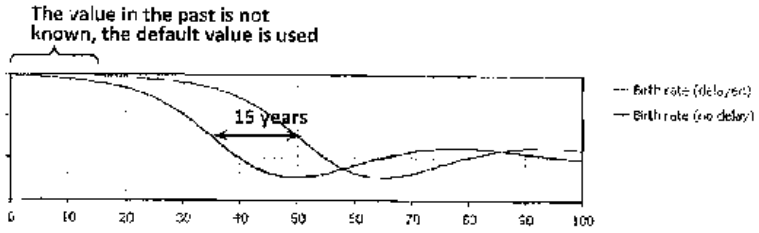


Figure 5.21 The birth rate with and without delay

Specifying units and performing unit checking

We will start with the **Population** stock. The stock contains individuals, so its unit is **individual** (this will be a new custom unit in our model). The **Births** and **Deaths** flows add and remove individuals to/from the stock, and are therefore measured in number of individuals per time unit. Their units are **individual / time**, where time is a pre-defined unit in AnyLogic.

This holds for all stocks and flows: a flow to/from a stock is always measured in ***<stock unit>*** **/ time** because a flow is a part of the stock derivative, i.e. d(Stock) / dt.

The variable **AverageLifetime** and the parameter **MaxAverageLifetime** are naturally measured in **time** units. Given that the unit of **Deaths** is **individual / time**, this is consistent with the formula for **Deaths**: **Population / AverageLifetime**.

Consider the formula for **AverageLifetime**:

AverageLifetime = MaxAverageLifetime * EffectOfCrowdingOnLifetime(Crowding)

What are the units of the table function **EffectOfCrowdingOnLifetime()**? The meaning of the function is to provide a coefficient reducing **MaxAverageLifetime** to a lower value that is currently applicable. The coefficient does not change the units of measurement, so the table function is unitless.

CarryingCapacity is the maximum population that can be supported by the environment, so its unit of measurement is **individual**. **Crowding** is unitless; it is the fraction of the carrying capacity currently occupied.

MaturationDelay is naturally measured in **time** units. The only two variables that may seem to have not so intuitive units are the **MaxBirthRate** and **BirthRate**. At first glance, variables with such names should be measured in **individual / time**. However, **BirthRate** does not flow directly into the **Population** stock: in the formula of the **Births** flow it is a multiplier to population meaning "births per individual per time unit". Therefore, the units for **BirthRate** (and consequently for **MaxBirthRate**) are **1 / time**.

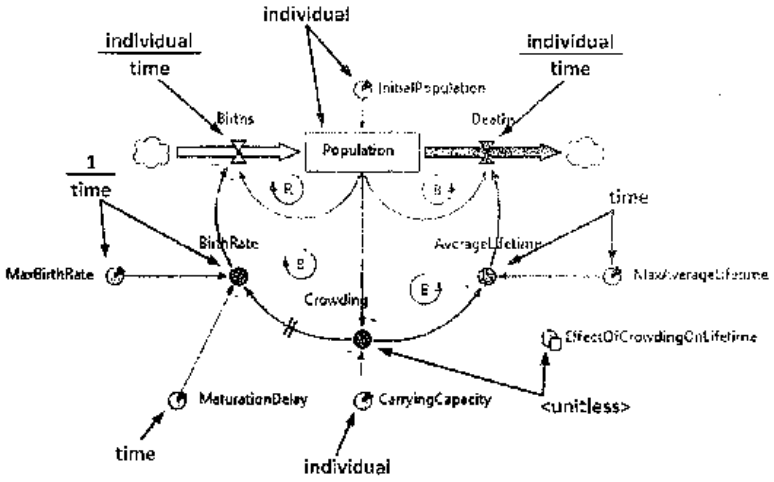


Figure 5.22 Units in the population model

5.4. Other types of experiments. Interactive games

Example 5.1: New product diffusion - compare runs

To show how to perform compare runs and sensitivity analysis experiments with the system dynamics models, we will use a simple model of Example 2.1:, "New product diffusion". The stock and flow diagram for that model is shown in Figure 2.2. Briefly: a company starts selling a new product in the market of a known constant size. Consumers are sensitive to advertising and word-of-mouth. The product has an unlimited lifetime and does not generate repeated purchases. A consumer needs only one product. The model forecasts the sales dynamics.

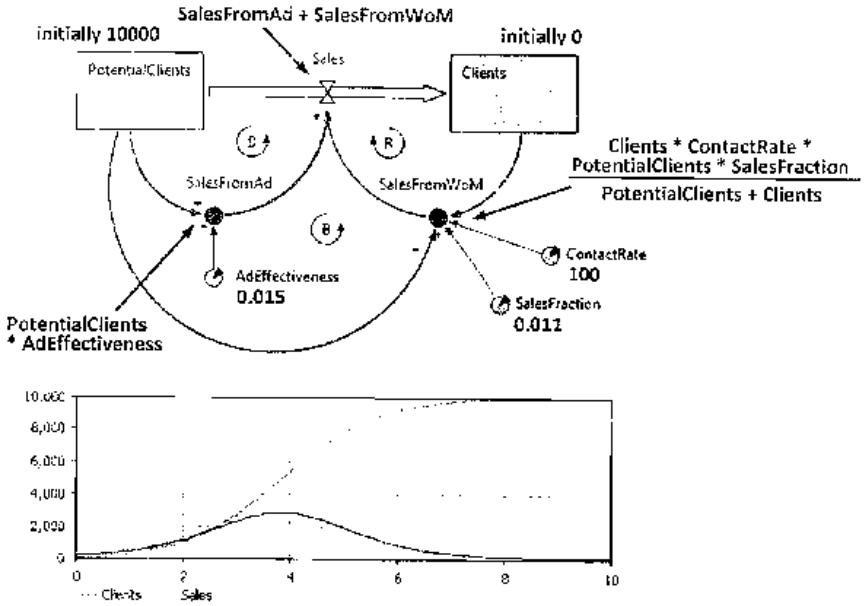


Figure 5.23 The model of new product diffusion and its behavior with the default parameters values

The model exhibits the S-shaped growth caused by the initial dominance of the reinforcing word of mouth feedback, gradually taken over by the balancing market saturation feedback.

First, we will build an experiment that allows us to manually vary the **ContactRate** parameter and compare the model behavior. Namely, we will compare the growth of the client base and the sales rate.

▶ **Create datasets for the key variables:**

1. Right-click the **Clients** stock and choose **Create data set** from the context menu. Drag the dataset to the right of the diagram.
2. In the dataset properties, select **Update data automatically** with **Recurrence time** 0.1. Given that the simulated period is 10 time units, we will have 100 samples.
3. Do the same for the **Sales** flow.

Now you have the datasets that will contain the history of the key variables at the end of a simulation run. The next step is to create a new experiment.

▶ **Create a compare runs experiment:**

4. Right-click the model (top-level) item in the **Projects** tree and choose **New | Experiment** from the context menu.
5. In the first page of the wizard, choose the **Compare runs** experiment type and click **Next**.
6. In the **Parameters** page, add the **ContactRate** parameter to the selected set and proceed to the next page.
7. In the **Charts** page of the wizard, setup the two output charts – one for **Clients** and one for **Sales**. To do this, select **dataset** in the first table column, type the names of the charts in the second column, and enter the references to the dataset you have defined in **Main**, see **Figure 5.24**. Click **Finish**.

In the last page of the wizard, you were to specify the scalar values or datasets in the model that are considered as the model output to be compared. As **Main** is the root (top level object) in the model, the syntax is **root.ClientsDS**.

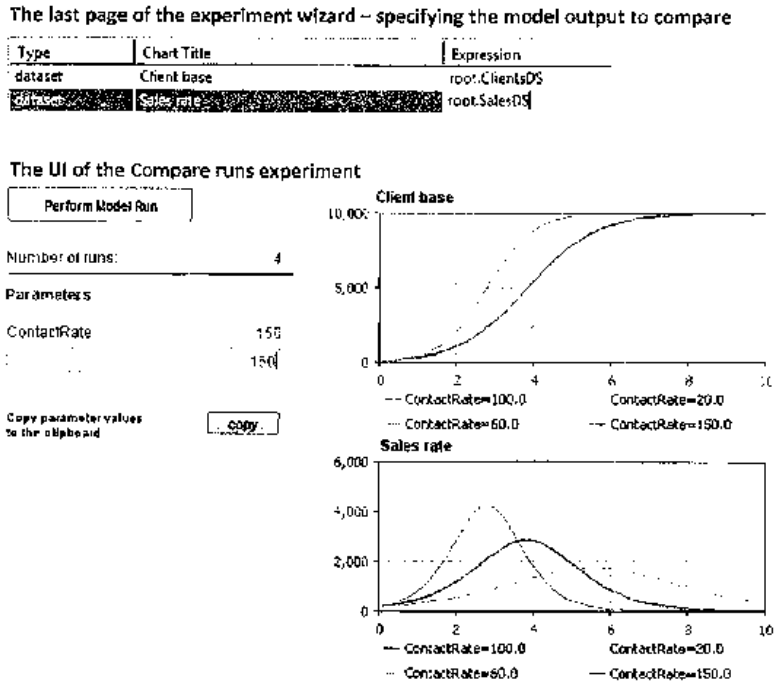


Figure 5.24 Compare runs experiment

As the wizard finishes, a new experiment is created. Its editor opens, and you can see a new item in the **Projects** tree. You can adjust the charts layout and the window size.

▶ Run the **Compare runs** experiment:

8. Right-click the **CompareRuns** experiment in the **Projects** tree and choose **Run** from the context menu.
9. Click the button **Perform model run** to obtain the simulation results for the default value of the **ContactRate** parameter.
10. Change the parameter value and perform a couple more runs.

Each curve in a chart corresponds to a particular simulation run. You can copy the datasets by clicking in the legend and selecting **Copy all** or **Copy selected** from the context menu. The last set of values to the parameters being varied can be copied to the clipboard with the **Copy** button.

You can further customize the interface of the experiment. For example, you may wish to view the value of the parameter in each run in the chart legend. To do this:

11. Open the **Advanced** page of the experiment properties and find the **After simulation run** code section.
12. In both function calls – `chart0.addDataSet(...` and `chart1.addDataSet(...` – replace the argument `"Run " + numberOfRuns` with `"ContactRate=" + root.ContactRate`.
13. Run the experiment again.

Example 5.2: New product diffusion - sensitivity analysis

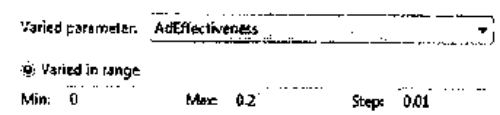
Our next experiment type is sensitivity analysis. We will explore how sensitive our model is to advertising effectiveness. We will assume you have the model with the two datasets (**ClientsDS** and **SalesDS**) already created.

▶ **Create a sensitivity analysis experiment:**

1. Right-click the model (top-level) item in the **Projects** tree and choose **New | Experiment** from the context menu.
2. In the first page of the wizard, choose the **Sensitivity analysis** experiment type and click **Next**.
3. In the **Parameters** page, select the **AdEffectiveness** parameter and specify that it is varied from 0 to 0.2 with the step of 0.01, see Figure 5.25. Proceed to the next page.
4. In the **Charts** page of the wizard setup the two output charts – one for **Clients** and one for **Sales** – in the same way you did for the compare runs experiment (see Figure 5.24). Click **Finish**.
5. Right-click the **SensitivityAnalysis** experiment in the **Projects** tree and choose **Run** from the context menu.

6. Click the **Run** button and view the charts.

AnyLogic performs a series of runs varying the **AdEffectiveness** parameter. The simulation output is added to the charts after each run, and the chart legend displays the parameter values.



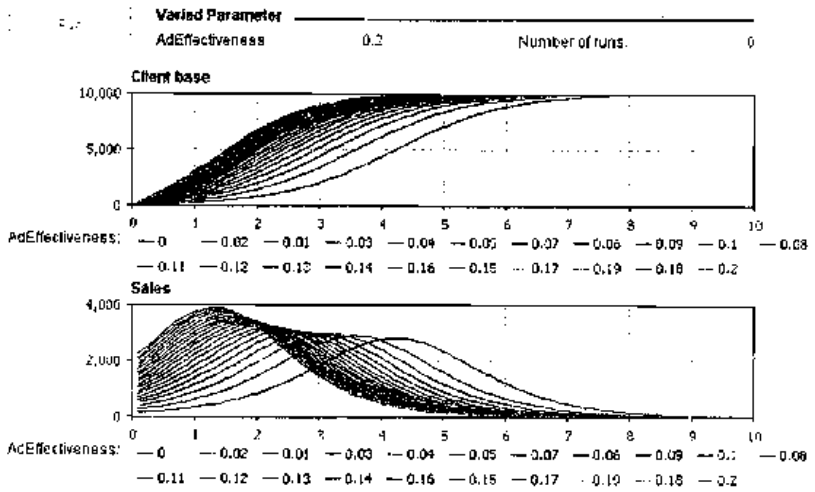


Figure 5.25 Sensitivity analysis experiment

Example 5.3: Epidemic model – calibration

For the next two examples we will use another model – a classic epidemic model *SIR*, or Susceptible Infectious Recovered ("Compartmental models in epidemiology", n.d.), see Figure 5.26. We are modeling a spread of contagious disease in a population where everybody is susceptible and, having been infected, can transmit the disease to other individuals. The disease has a finite duration and, after recovering, an individual becomes immune. The model has two parameters we will focus on:

- **Infectivity** – the probability of disease transmission during contact of an infected individual with a susceptible individual.

- **AverageIllnessDuration** – the average period an individual remains infectious after being infected.

These parameters cannot be measured (at least with reasonable efforts) directly in the real world, in contrast with **TotalPopulation** (which we assume we know) and **ContactRate** (which we assume we can measure). Before we use the model, we need to find out the values of the immeasurable parameters. One way of doing that is to *calibrate* the model – use historical data of a similar case and adjust the parameter values so that the model output reproduces the historical data as closely as possible.

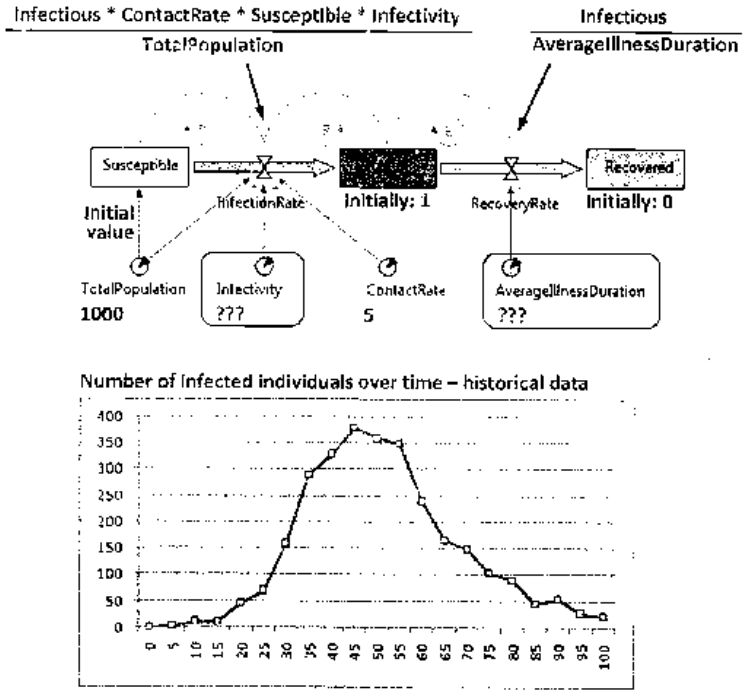


Figure 5.26 The SIR epidemic model and the historical data

The historical data is shown in Figure 5.26 as the number of infected people over time in a population of a thousand people. To calibrate the model, we will use the OptQuest optimizer built into AnyLogic, which will iteratively run the model, compare its output with the historical curve, change the parameter values, perform another run, etc., until it decides it has done its best (or the iteration limit is reached).

▶ **Create the SD model:**

1. Create the SIR model, as shown in Figure 5.26. Use some values for Infectivity and AverageIllnessDuration, say 0.08 and 12.
2. In the Model time property page of the default simulation experiment (the one that has been created automatically), set **Stop to Stop at the specified time**: 100.
3. Run the model (run the default simulation experiment) to make sure it works as expected and the dynamics of **Infectious** stock is bell-shaped.

Before we do the calibration, we need to create a dataset in the model that will keep the model output at the end of a simulation run.

4. Right-click the **Infectious** stock and choose **Create data set** from the context menu. A dataset **InfectiousDS** is created.
5. In the dataset properties, switch the update mode to **Update data automatically**.

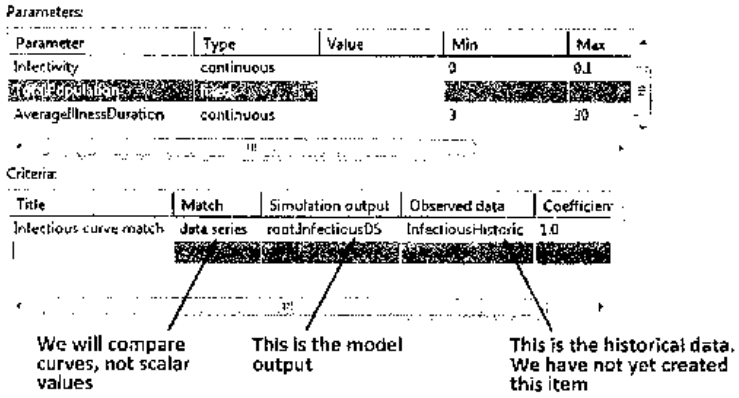


Figure 5.27 Settings of the calibration experiment

▶ **Create a calibration experiment:**

6. Right-click the model (top-level) item in the **Projects** tree and choose **New | Experiment** from the context menu.
7. In the first page of the wizard, choose the **Calibration** experiment type and click **Next**.
8. In the **Parameters and Criteria** page of the wizard, in the **Parameters** table change the **Type** of the parameters we are calibrating from fixed to continuous. Set the range of the **Infectivity** to 0 .. 0.1 and the range of **AverageIllnessDuration** to 3 .. 30.
9. In the **Criteria** table, fill in one row, as shown in Figure 5.27. Click **Finish**.

The calibration experiment with the default UI has been created. If you compile the model, you will get the error "InfectiousHistoric cannot be resolved.". This is because we have not created the item in the model that contains the historical data, although we have already indicated that this item will be used.

▶ Create a table function and import the historical data:

10. Open the editor of the Calibration experiment and drag the Table function from the System Dynamics palette to the editor. Change the name of the function to InfectiousHistoric.
11. Import the historical data into the table function. If you have the data in a spreadsheet, text editor, or another application that can copy the data on the clipboard in e.g. tab-separated format, and click the Paste from clipboard button on the right of the table in the table function properties. Leave the default Interpolation and Out of range settings.
12. Run the Calibration experiment and click the Run calibration button. Watch the progress of calibration. It is beautiful!

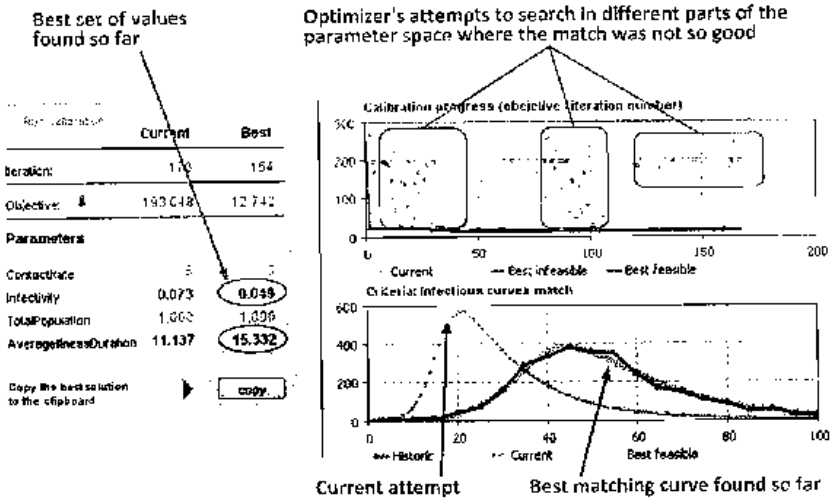


Figure 5.28 Calibration experiment in progress (iteration 170 of maximum 500)

The values of Infectivity and AverageIllnessDuration found in the calibration experiment are about 0.049 and 15.33. You can copy the values from the calibration experiment (see the Copy button in Figure 5.28) and paste them to the simulation or any other experiment by clicking Paste form clipboard in the General page of the experiment properties.

Example 5.4: Epidemic model - instant charts

A typical scenario in system dynamics modeling is the instant re-simulation of the model as the user changes the parameters and the output charts immediately reflecting the change. While this scenario is planned to be included in the future versions of AnyLogic as a standard one, you can create such a scenario in the current version on the basis of the standard experiment compare runs.

We will expose the three parameters of the model (**Infectivity**, **ContactRate**, and **AverageIllnessDuration**) to the user as three sliders in the experiment window. The chart in the same window will show the simulation results immediately after the user has changed the parameters.

▶ **Prepare the model and create a compare runs experiment:**

1. In the **Main** object of the Epidemic model, add datasets for all stocks. Do it in the same way we did it for the Infectious stock in the previous example. You should get **SusceptibleDS**, **InfectiousDS**, and **RecoveredDS**.
2. In the properties of the datasets, select **Update data automatically**.
3. In the **Editor** property page of the parameters **Infectivity**, **ContactRate** and **AverageIllnessDuration**, choose the **Slider** editor type. Set the slider ranges:
   for **Infectivity**: from 0.01 to 0.20
   for **ContactRate**: from 1 to 20
   for **AverageIllnessDuration**: from 3 to 30.
4. Right-click the model item in the **Projects** tree and choose **New | Experiment** from the context menu. In the wizard choose the **Compare Runs** experiment type, change its name to **InstantCharts**, and click **Next**.
5. In the **Parameters** page of the wizard, choose the three parameters you are going to vary (all except for **TotalPopulation**) and move them to the **Selection** list. Click **Next** and, on the next page, click **Finish**. A new experiment is created, and you can see its default UI with three sliders, see Figure 5.29.
6. Delete the button and some unnecessary graphics, as shown in the same Figure 5.29. You will not need them.
7. In the **InstantCharts** experiment, create three datasets with the same names as the datasets in the **Main** object: **SusceptibleDS**, **InfectiousDS**, and **RecoveredDS**. In the datasets properties, deselect the option **Use run number as horizontal axis value**. These three datasets will be used to copy the data from the latest simulation run.
8. Create a plot (drag the **Plot** object from the **Analysis** palette) and add to it the three datasets. Adjust the line colors as you wish, see Figure 5.29.

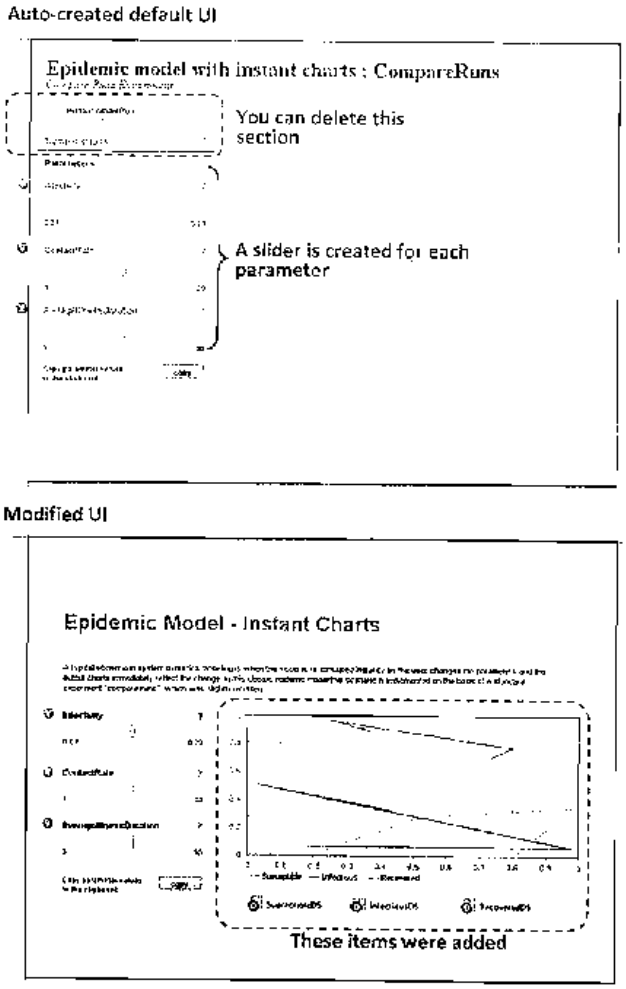

Auto-created default UI
Epidemic model with instant charts : CompareRuns
You can delete this section
A slider is created for each parameter
Modified UI
Epidemic Model - Instant Charts
These items were added

Figure 5.29 Auto-created and modified UI of the compare runs experiment

Now we have the controls to vary the parameters and the chart that is ready to display the simulation results. To complete the model, we only need to:

- Modify the slider actions so that each time the slider is moved, a new simulation run is performed.
- Upon completion of a simulation run, re-fill the datasets displayed in the plot with the most recent model output.

▶ **Set up the slider actions and add an action executed after each simulation run:**

9. Add the function call **run();** to the action of each slider.
10. Open the **Advanced** page of the **InstantCharts** experiment and type this code in the **After simulation run** field:

```
SusceptibleDS.fillFrom( root.SusceptibleDS );
InfectiousDS.fillFrom( root.InfectiousDS );
RecoveredDS.fillFrom( root.RecoveredDS );
```

This code deletes the current contents of the datasets at the experiment level and copies there the content of the datasets in the model (**root** refers to the top-level object of the model, **Main**).

11. Run the **InstantCharts** experiment and use the sliders to change the parameters. The plot instantly displays the simulation output.

Note that the **run()** function in the compare runs experiment performs a simulation run in the fastest possible mode with animation turned off.

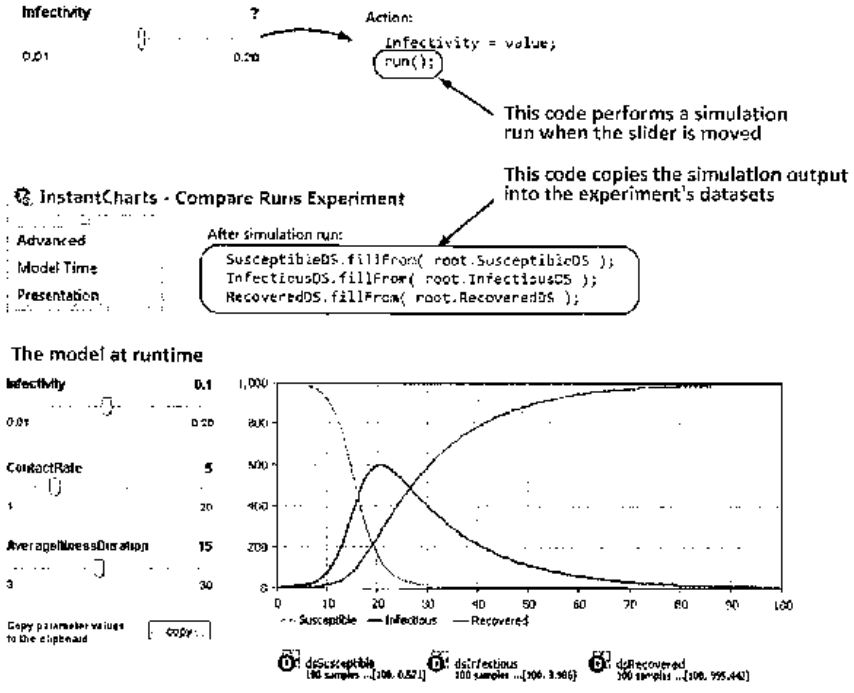


Figure 5.30 The last steps in experiment setup. The completed model at runtime

Example 5.5: Stock management game

We will now build a *game*, or a *flight simulator* – an interactive model that exposes its output to the user as the simulation is being performed and allows the user to make decisions; for example, to change parameters in a running model and observe the effect.

Typically, such games work in step-pause mode: the model executes for a certain period, then it is put in the paused state, at which time the user observes the output, makes decisions, and starts the simulation for the next period. This may make sense when the real system is also controlled only at certain regular points in time – for example, each month or each quarter. In addition to that scenario, AnyLogic allows you to build games where the model executes in a scale to real time, and the user is able to make changes continuously, at arbitrary moments. We will first implement the latter scenario, and then add steps and pauses.

The model will be a very simple supply chain, see Figure 5.31. The user is to manage the stock by controlling the order rate. The sales rate is exogenous and will be modeled as randomly changing.

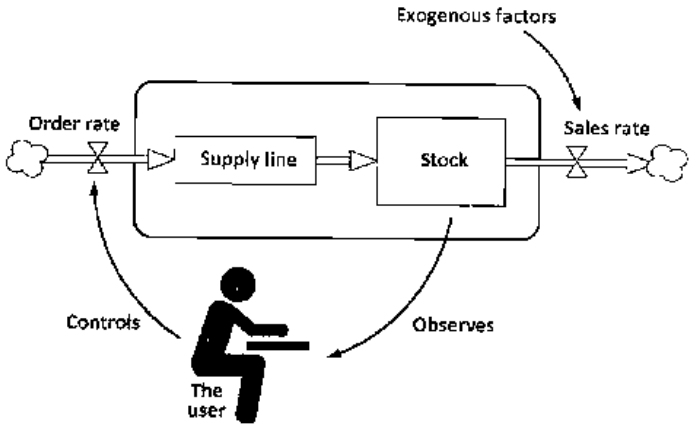


Figure 5.31 Architecture of the stock management game

▶ **Create the system dynamics model:**

1. Create the stock and flow diagram, as shown in Figure 5.32.

Note that both **OrderRate** and **Demand** are marked as constants (see Section 5.2): therefore, they only have initial values, and no equations. These two "variables" will be controlled from outside the system dynamics model: **Demand** will be changed by a recurrent discrete event, and **OrderRate** will be controlled by the user.

The equation of **SalesRate** uses the conditional operator (see Section 10.5). While there is product in stock, it sells at the **Demand** rate, otherwise nothing is sold.

2. Add the event ExogenousDemandChange and set up it to occur at every time unit. In the **Action** field of the event, type the following code:

```
Demand = max( 0, Demand + uniform( -1, 1 ) );
```

This code increases or decreases the value of the **Demand** variable by a random amount, uniformly distributed between -1 and 1. The **max()** function protects **Demand** from falling below zero.

Do not confuse time unit (see Section 16.1) and time step. The time unit is the unit of simulated time and can correspond to an hour, day, week, etc. This correspondence is established in the properties of the model. The time step is a micro-step of numerical methods in the AnyLogic simulation engine. It is set up in the **Advanced** properties of the experiment and is typically much smaller than the time unit (say, 0.001).

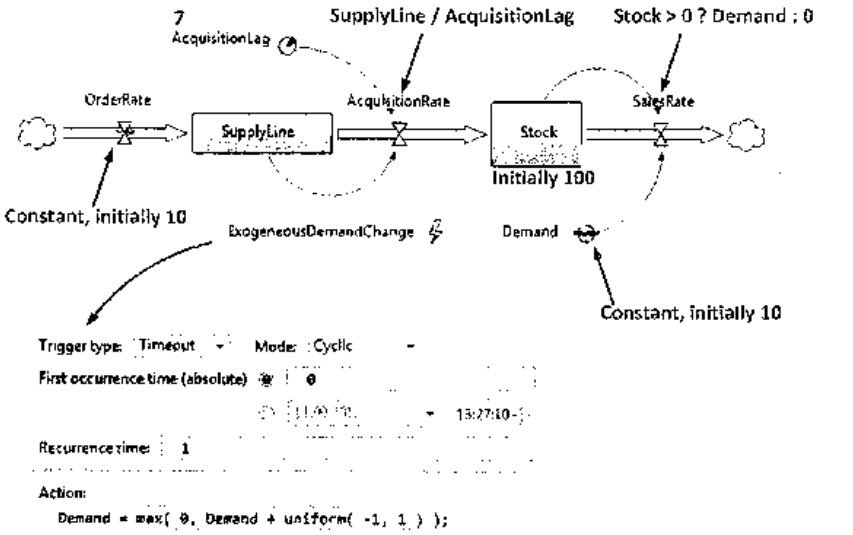


Figure 5.32 The stock management model without the controlling feedback

3. Add a **Time stack chart** from the **Analysis** palette. Add there just one data item with the expression **max(0, Stock)** (**Stock** still can fall slightly below zero due to numeric errors.). Set the chart **Time window** to 1,000 and the number of samples to display to 1,000 as well.

4. Select the **Simulation** experiment in the **Projects** tree and open the **Presentation** page of its properties. Set the **Execution mode** to **Real time with scale** 50. This is a better speed for our game.
5. Run the model and see how the stock changes over time.

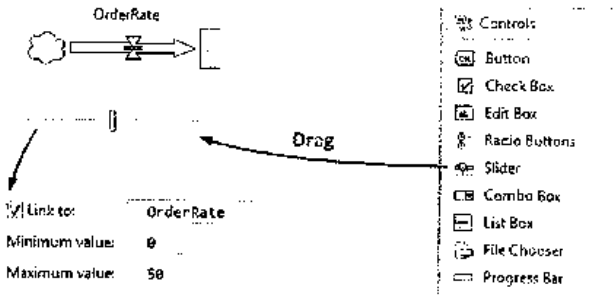


Figure 5.33 Slider – an element of the user interface

▶ **Close the control loop: create a slider that varies the order rate:**

6. Drag a **Slider** from the **Controls** palette and link it to the **OrderRate** variable. Set the slider range to 0..50, see Figure 5.33.

If you link a slider to a dynamic variable *with an equation* (the one that is not marked as constant), the slider's attempts to change the variable will be unsuccessful – the equation will immediately override the value assigned by the slider.

7. Run the model and try to keep the stock within a certain interval – say, between 100 and 500.

In our current setup, the user is able to see the entire stock and flow diagram with all of the variable values. This is not always desirable, and it is up to the game designer to decide which variables are to be exposed to the user. In the next step, we will create a separate interface page that will contain only the selected model output and the control elements.

▶ **Create a separate interface page using view area**

8. Drag a **View area** from the **Presentation** palette and place it at (0, 600). This will be the top left corner of the user screen. Give the view area the name **viewUserScreen** and the title "User screen".

As the model window size is 800 by 600 pixels (these settings can be checked and changed in the **Window** page of the experiment properties), the Y-coordinate of the view area, 600, ensures the user screen does not intersect with the model screen.

9. Move the chart and the slider to the user screen, i.e. below the Y coordinate 600.
10. In the slider properties, click the button **Add labels**.
11. In the **General** page of the Main object properties, write the following code in the **Startup code** field:
    viewUserScreen.navigateTo();
    This will display the user screen right at the beginning of the simulation.
12. In the **Window** page of the **Simulation** experiment properties, deselect the checkboxes **Enable panning** and **Enable zoom**. This will prevent the user from occasionally moving the canvas.
13. Run the model. Now you can only see the stock time chart and the slider.

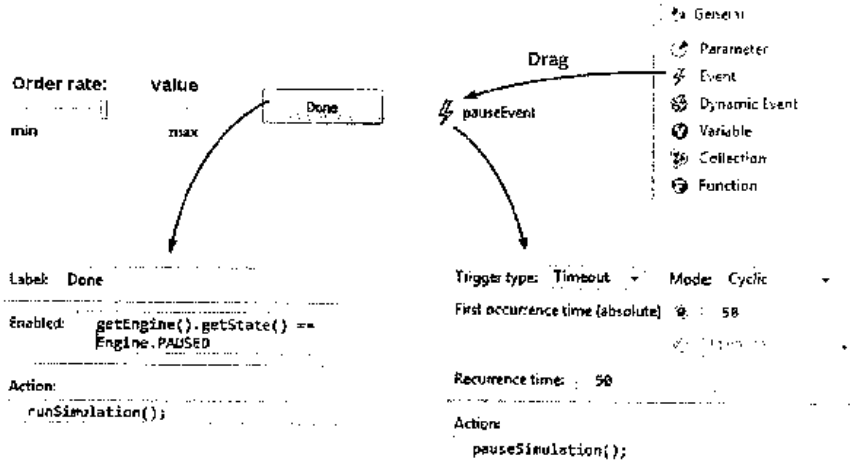
You can still switch to the model screen by clicking the **Navigate to view area** toolbar button and choosing **[Origin]**. To disable that toolbar section, go to the **Window** page of the experiment properties and deselect **Model navigation** and **Make toolbar customizable at runtime** checkboxes.

As a final phase of this exercise, we will implement the step-pause game mode. The step duration will be 50 time units (days). To schedule the pause at the end of a 50-day period, we will use a cyclic event (see Section 8.2) and the AnyLogic engine function **pauseSimulation()**. To indicate that the user has finished making decisions and to run the simulation again, we will use a button and the function **runSimulation()**.

▶ **Implement the step-pause game mode:**

14. Open the **Presentation** page of the **Simulation** experiment properties and change the **Execution mode** to **Virtual time**. The simulation will now be performed as fast as possible. (If you want the charts to progress with some animation effects, you can stay within the real time mode with a higher scale.)
15. Open the editor of the **Main** object and drag a **Button** from the **Controls** palette. Place the button on the user screen and change its label to "Done." Type the following condition in the **Enabled** field of the button:
    getEngine().getState() == Engine.PAUSED
    and write this function call in the **Action** field:
    runSimulation();
16. Copy the condition in the **Enabled** field of the button and paste it to the **Enabled** field of the slider. These conditions ensure that the user will only be able to change the order rate when the model is in the paused state.
17. Drag the **Event** object from the **General** palette. Set the event mode to Cyclic, the **First occurrence time** to 50, and the **Recurrence time** to 50. In the **Action** field of the event write: **pauseSimulation();**

18. Run the model. The model now works in step-pause mode, allowing the user to change the order rate once every 50 days.



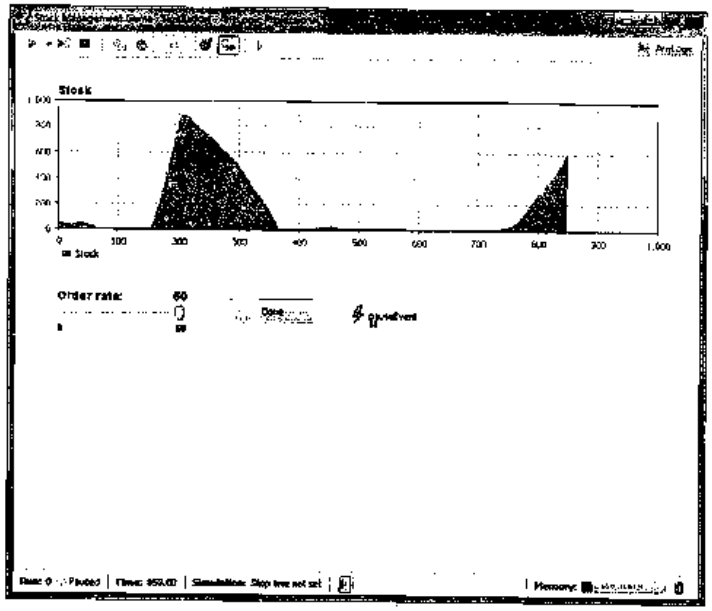


Figure 5.34 Implementation of the step-pause game mode

5.5. Exporting the model and publishing it on the web

AnyLogic models are 100% Java. The simulation engine, the numerical methods, the optimizer, the animation, the user interface, and the model itself – all that is in Java, can be separated from the model development environment, runs on any platform, and can be published on the web. You can deliver the model to your client, share it with colleagues, or make it publicly available in the form of a Java applet. End users will be able to run the model directly in their browsers without installing any software. Figure 5.35 explains the export options.

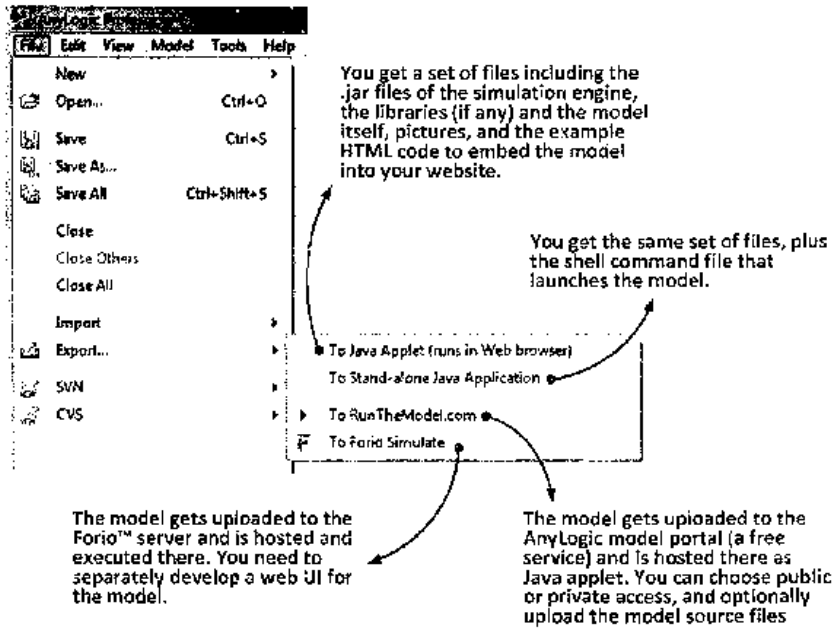

File Edit View Model Tools Help
New
Open... Ctrl+O
Save Ctrl+S
Save As...
Save All Ctrl+Shift+S
Close
Close Others
Close All
Import
Export...
SVN
CVS
You get a set of files including the .jar files of the simulation engine, the libraries (if any) and the model itself, pictures, and the example HTML code to embed the model into your website.
You get the same set of files, plus the shell command file that launches the model.
To Java Applet (runs in Web browser)
To Stand-alone Java Application
To RunTheModel.com
To Forio Simulate
The model gets uploaded to the Forio™ server and is hosted and executed there. You need to separately develop a web UI for the model.
The model gets uploaded to the AnyLogic model portal (a free service) and is hosted there as Java applet. You can choose public or private access, and optionally upload the model source files

Figure 5.35 AnyLogic model export options

The easiest way to share the model is to export it to a free AnyLogic model portal, RunTheModel.com. We will export Example 5.5: "Stock management game".

▶ **Export the model to RunTheModel.com:**

1. Right-click the model item in the **Projects** tree and choose **Export| To RunTheModel.com** from the context menu.
2. On the next page, you can select the experiment to export (in case multiple experiments were created), and the language.
3. The next page is the login page. If you do not have a RunTheModel.com account yet, you can create one by following the link on the page.

4. The next page asks whether this will be a new model or an update/new version of a previously uploaded model.

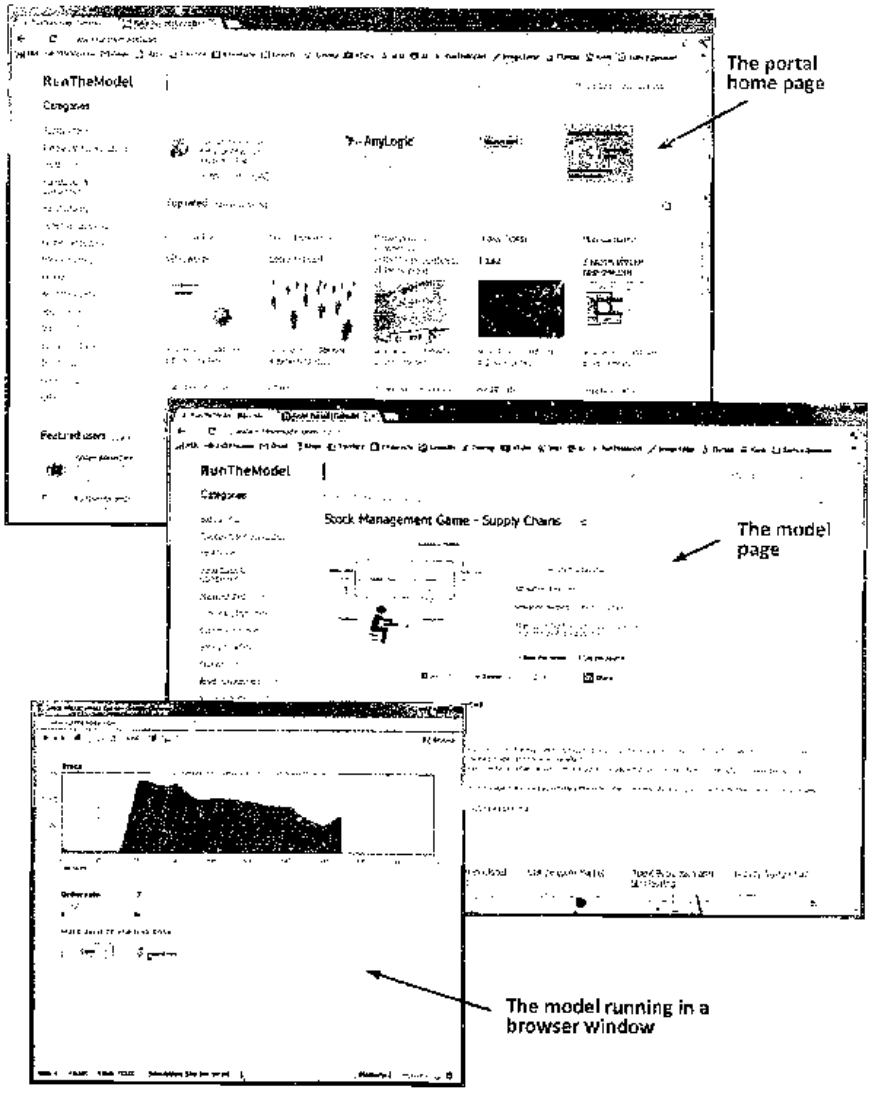


Figure 5.36 The Stock Management Game exported to RunTheModel.com

5. The next page allows you to modify or write the model description (by default, the description is taken from the **Description** field of the model),

choose the icon, set the access rights, and say whether you want the model source files to be available to download for those who will view the model page.

In case you choose the **Private** access type, the model will not be listed in the portal directory, and you will get a special link to the model page, which you can share with selected users.

6. On the next page you specify category, application area, modeling method, and tags. Tags are needed to improve searching among the uploaded models. For example, for our model the tags could be "game, flight simulator, system dynamics, interactive models, supply chain".
7. The next page allows you to check which source files get uploaded (if you have chosen to upload them). By clicking **Next** you start the export process.
8. When the export is finished, the direct link to your model is displayed on the last page of the wizard. Follow the link.

Chapter 6. Multi-method modeling

The three modeling methods, or paradigms (see Chapter 2), are essentially the three different viewpoints the modeler can take when mapping the real world system to its image in the world of models. The system dynamics paradigm suggests to abstract away from individual objects, think in terms of aggregates (stocks, flows), and the feedback loops. The discrete event modeling adopts a process-oriented approach: the dynamics of the system are represented as a sequence of operations performed over entities. In an agent based model the modeler describes the system from the point of view of individual objects that may interact with each other and with the environment.

Depending on the simulation project goals, the available data, and the nature of the system being modeled, different problems may call for different methods. Also, sometimes it is not clear at the beginning of the project which abstraction level and which method should be used. The modeler may start with, say, a highly abstract system dynamics model and switch later on to a more detailed discrete event model. Or, if the system is heterogeneous, the different components may be best described by using different methods. For example in the model of a supply chain that delivers goods to a consumer market the market may be described in system dynamics terms, the retailers, distributors, and producers may be modeled as agents, and the operations inside those supply chain components – as process flowcharts.

- **Frequently, the problem cannot completely conform to one modeling paradigm. Using a traditional single-method tool, the modeler inevitably either starts using workarounds (unnatural and cumbersome language constructs), or just leaves part of the problem outside the scope of the model (treats it as exogenous). If our goal is to capture business, economic, and social systems in their interaction, this becomes a serious limitation.**

AnyLogic meets this challenge by supporting all three modeling methods on a single modern object-oriented platform. With AnyLogic, a modeler can choose from a wide range of abstraction levels, can efficiently vary them while working on the model, and can combine different methods in one model.

In this chapter we offer an overview of most used multi-method model architectures, discuss the technical aspects of linking different methods within one model, and consider examples of multi-method models.

6.1. Architectures

The number of possible multi-method model architectures is infinite, and many are used in practice. Popular examples are shown in Figure 6.1. We will briefly discuss the problems where these architectures may be useful.

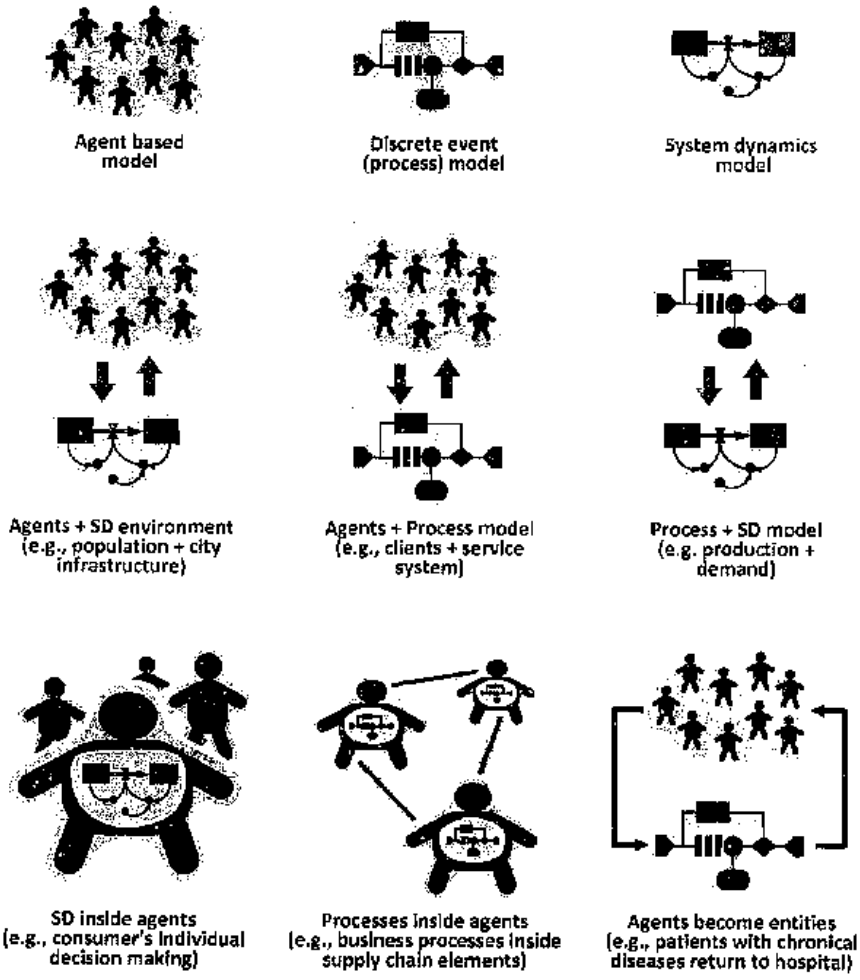

Agent based model
Discrete event (process) model
System dynamics model
Agents + SD environment (e.g., population + city infrastructure)
Agents + Process model (e.g., clients + service system)
Process + SD model (e.g. production + demand)
SD inside agents (e.g., consumer's individual decision making)
Processes inside agents (e.g., business processes inside supply chain elements)
Agents become entities (e.g., patients with chronical diseases return to hospital)

Figure 6.1 Popular multi-method model architectures

Agents in an SD environment. Think of a demographic model of a city. People work, go to school, own or rent homes, have families, and so on. Different

neighborhoods have different levels of comfort, including infrastructure and ecology, cost of housing, and jobs. People may choose whether to stay or move to a different part of the city, or move out of the city altogether. People are modeled as agents. The dynamics of the city neighborhoods may be modeled in system dynamics way, for example, the home prices and the overall attractiveness of the neighborhood may depend on crowding, and so on. In such a model agents' decisions depend on the values of the system dynamics variables, and agents, in turn, affect other variables.

The same architecture is used to model the interaction of public policies (SD) with people (agents). Examples: a government effort to reduce the number of insurgents in the society; policies related to drug users or alcoholics.

Agents interacting with a process model. Think of a business where the service system is one of the essential components. It may be a call center, a set of offices, a Web server, or an IT infrastructure. As the client base grows, the system load increases. Clients who have different profiles and histories use the system in different ways, and their future behavior depends on the response. For example, low-quality service may lead to repeated requests, and, as a result, frustrated clients may stop being clients. The service system is naturally modeled in a discrete event style as a process flowchart where requests are the entities and operators, tellers, specialists, and servers are the resources. The clients who interact with the system are the agents who have individual usage patterns.

Note that in the previous example the agents can be created directly from the company CRM database and acquire the properties of the real clients. This also applies to the modeling of the company's HR dynamics. You can create an agent for every real employee of the company and place them in the SD environment that describes the company's integral characteristics (the first architecture type).

A process model linked to a system dynamics model. The SD aspect can be used to model the change in the external conditions for an established and ongoing process: demand variation, raw material pricing, skill level, productivity, and other properties of the people who are part of the process.

The same architecture may be used to model manufacturing processes where part of the process is best described by continuous time equations – for example, tanks and pipes, or a large number of small pieces that are better modeled as quantities rather than as individual entities. Typically, however, the rates (time derivatives of stocks) in such systems are piecewise constants, so simulation can be done analytically, without invoking numerical methods.

System dynamics inside agents. Think of a consumer market model where consumers are modeled individually as agents, and the dynamics of consumer decision making is modeled using the system dynamics approach. Stocks may represent the consumer perception of products, individual awareness, knowledge, experience, and so on. Communication between the consumers is modeled as discrete events of information exchange.

A larger-scale example is interaction of organizations (agents) whose internal dynamics are modeled as stock and flow diagrams.

Processes inside agents. This is widely used in supply chain modeling. Manufacturing and business processes, as well as the internal logistics of suppliers, producers, distributors and retailers are modeled using process flowcharts. Each element of the supply chain is at the same time an agent. Experience, memory, supplier choice, emerging network structures, orders and shipments are modeled at the agent level.

Agents temporarily act as entities in a process. Consider patients with chronic diseases who periodically need to receive treatment in a hospital (sometimes planned, sometimes because of acute phases). During treatment, the patients are modeled as entities in the process. After discharge from the hospital, they do not disappear from the model, but continue to exist as agents with their diseases continuing to progress until they are admitted to the hospital again. The event of admission and the type of treatment needed depend on the agent's condition. The treatment type and timeliness affect the future disease dynamics.

There are models where each entity is at the same time an agent exhibiting individual dynamics that continue while the entity is in the process, but are outside the process logic – for example, the sudden deterioration of a patient in a hospital.

The choice of model architecture and methods

AnyLogic, designed as a multi-method object-oriented tool, allows you to create model architectures of any type and complexity, including those previously mentioned. You can develop complex, simple, flat, hierarchical, replicated, static, or dynamically changing structures.

The choice of the model architecture depends on the problem you are solving. The model structure reflects the structure of the system being modeled – not literally, however, but as seen from the problem viewpoint. The choice of modeling method should be governed by the criterion of *naturalness*. Compact, minimalistic, clean, beautiful, easy to understand and explain – if the internal texture of your model is like that, then you chose the right method.

6.2. Technical aspect of combining modeling methods

In this section we will consider the techniques of linking different modeling methods in AnyLogic.

Examples 5.1 - 5.21: Combining modeling methods

The very first thing you should know is that all model elements of all methods, be they SD variables, statechart states, entities, process blocks, and even animation shapes or business charts exist in the "same namespace": any element is accessible from any other element by name (and, sometimes, "path" – the prefix describing the location of the element).

The following examples are all taken from the real projects and purged of all unnecessary details. This set, of course, does not cover everything, but it does give a good overview of how you can build interfaces between different methods.

System dynamics -> discrete elements

The system dynamics model is a set of continuously changing variables. All other elements in the model work in discrete time (all changes are associated with events). SD itself does not generate any events, so it cannot *actively* make an impact on agents, process flowcharts, or other discrete time constructs. The only way for the SD part of the model to affect a discrete element is to let that element watch on a condition over SD variables, or to use SD variables when making a decision. Figure 6.2 shows some possible constructs.

A and B. SD triggers an event or a statechart transition. Events (low-level constructs that allow scheduling a one-time or recurrent action, see Section 8.2) and statechart transitions (see Section 7.3)are frequent elements of agent behavior. Among other types of triggers, both can be triggered by a condition – a Boolean expression.

If the model contains dynamic variables, all conditions of events and statechart transitions are evaluated *at each integration step*, which ensures that the event or transition will occur exactly when the (continuously changing) condition becomes true.

In Figure 6.2 A and B the discrete elements are waiting for the **Money** stock to fall below zero, and for the **Interest** to rise higher than a given threshold value. The event and the statechart can be located on the same level as the SD, or in a different active object.

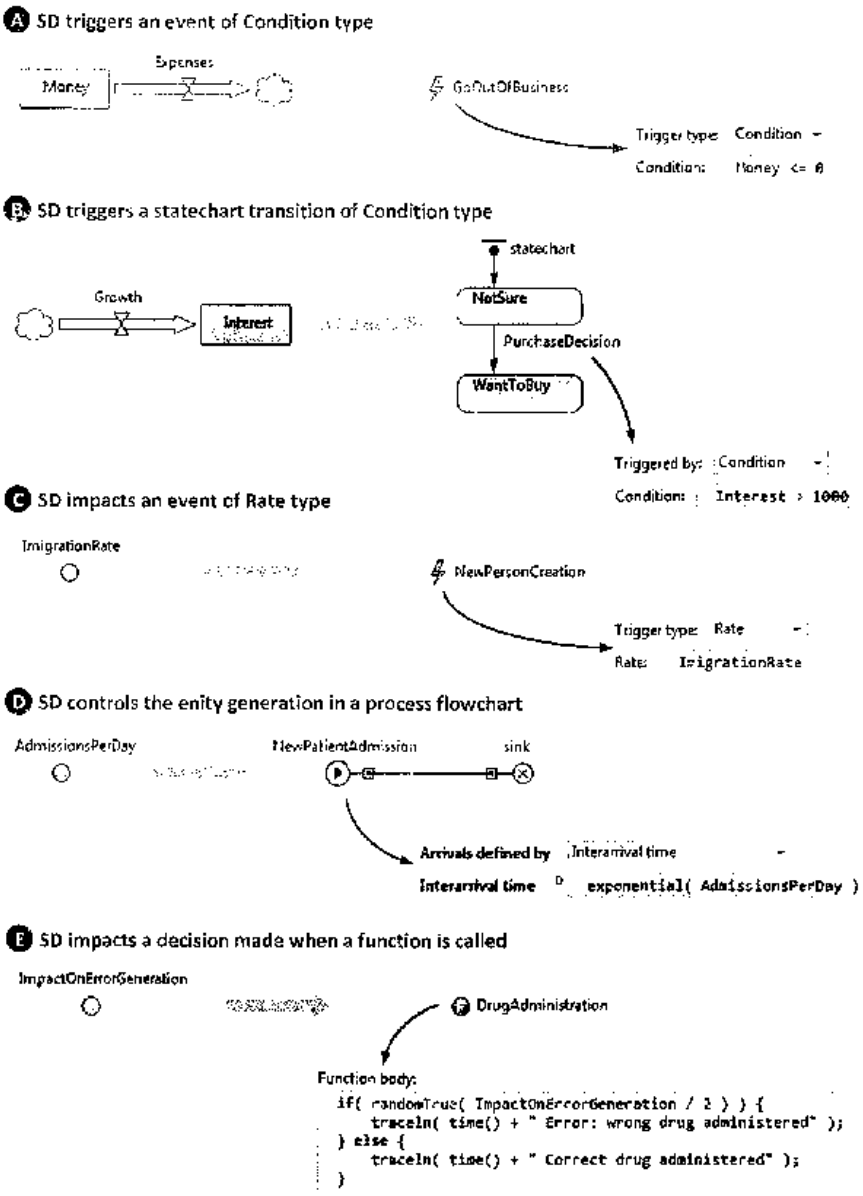


Figure 6.2 SD impacts discrete elements of the model

C and D. SD controls the rate of recurring discrete events. In the case of C there is an event (see Section 8.2) with a rate trigger type (the time between subsequent

occurrences is exponentially distributed). The event rate is set to the dynamic variable, and each subsequent occurrence is scheduled according to the current value of the variable.

In the case D the Source block **NewPatientAdmissions** generates new entities at the rate defined by the dynamic variable **AdmissionsPerDay**. The arrivals are defined in the form of interarrival time and not in the form of rate, because the rate is not re-evaluated during the simulation, whereas the interarrival time is re-evaluated after each new entity.

Note that if the value of the dynamic variable changes *in between* two subsequent event occurrences (or in between two entity arrivals), this will not be "noticed" immediately, but only at the next event occurrence (or next entity arrival).

E. The SD variable is used in a decision made in the DE part of the model. This example just shows that SD variables can be freely used in the actions and expressions inside the discrete elements of the model: conditions, functions, actions of events and transitions, **On enter/On exit** fields of process objects, and so on. Dynamic variables in that sense do not differ at all from plain (Java) variables.

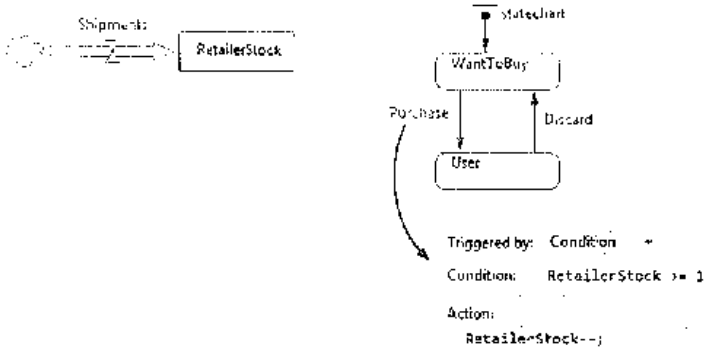
Discrete elements -> system dynamics

F. The SD stock triggers a statechart transition, which, in turn, modifies the stock value. Here, the interface between the SD and the statechart is implemented in the pair condition/action. In the state **WantToBuy**, the statechart tests if there are products in the retailer stock, and if there are, buys one and changes the state to **User**.

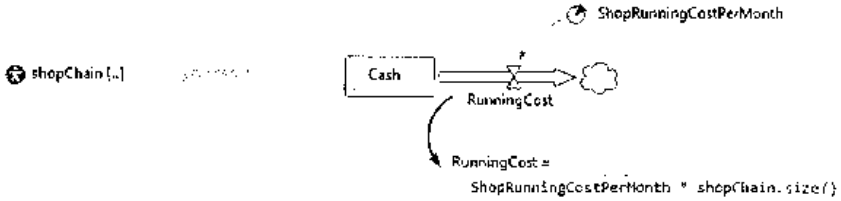
You can freely change the values of the system dynamic *stocks* from outside the system dynamics part of the model. This does not interfere with the differential equation solving: the integrator will just start with the new value. However, trying to change the value of a flow or auxiliary variable that has an equation associated with it, is not correct: the assigned value will be immediately overridden by the equation, so assignment will have no effect.

G. The size of the agent population is used in the SD equation. The equations in the system dynamics part of the model can reference not only the SD variables and functions, but also the arbitrary discrete elements in the model. The flow **RunningCost** out of the **Cash** stock depends on the current number of shops in the chain, and each shop is an agent. The function call **shopChain.size()** returns the number of agents in the population **shopChain**.

F The SD stock triggers a statechart transition, which, in turn, modifies the stock value



G Number of agents impacts an SD flow



H Statistics on agent population impact an SD variable [may be inefficient!]

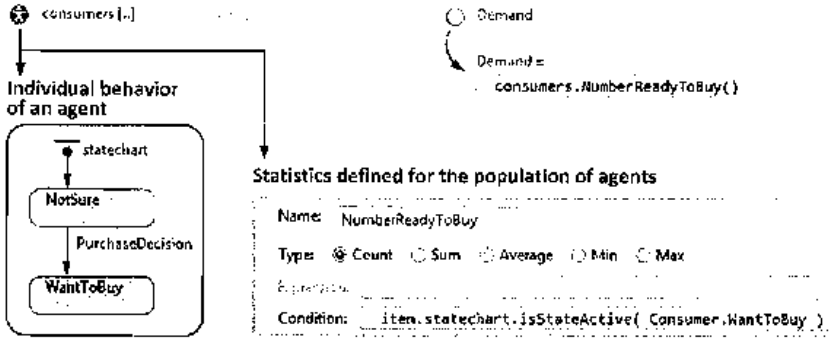


Figure 6.3 Discrete model elements impact SD (part 1)

H and I. The statistics on agent population impacts an SD variable. Here, we are calculating the number of agents in a population that are in a particular state **WantToBuy**, and provide that value to the SD variable **Demand**.

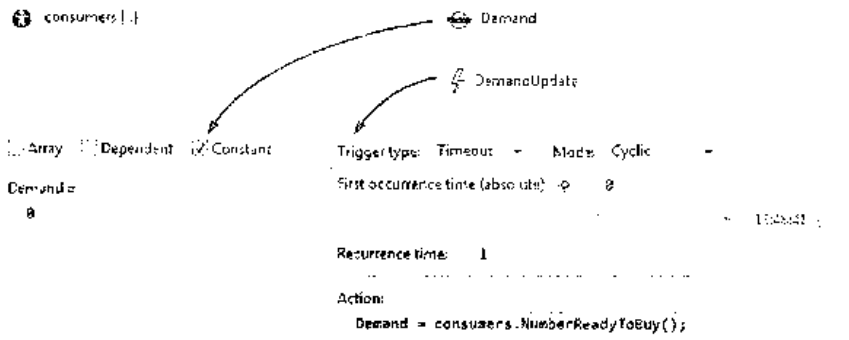
In the case H this is implemented in a straightforward way: the call to the **NumberReadyToBuy()** is typed directly into the equation field of **Demand**, and, therefore, statistics are re-evaluated at each integration step. If there are many agents in the model, this may be very time consuming and slow down the simulation. In case I the same functionality is implemented more efficiently: the variable **Demand** is declared as "constant" (no equation, see Section 5.2), and its value is periodically updated by the recurrent event **DemandUpdate**, whose frequency is significantly lower than the frequency of the numeric integrator.

J. The SD variable is controlled by the statechart state. Consider a model of a consumer who has a certain degree of interest in the product modeled by an "individual" SD stock **Interest** (located inside the consumer model). As the consumer is waiting for the product to become available, his interest decreases. This is captured by the equation of **LossDueToUnavailability** flow that contains the function call **statechart.isStateActive(Waiting)**. This function returns **true** if the statechart is in a given state, and **false** otherwise. In this model you also can implement a loop back from the SD to the statechart if you let the transition **InterestLost** occur when the value of the stock falls below a certain threshold.

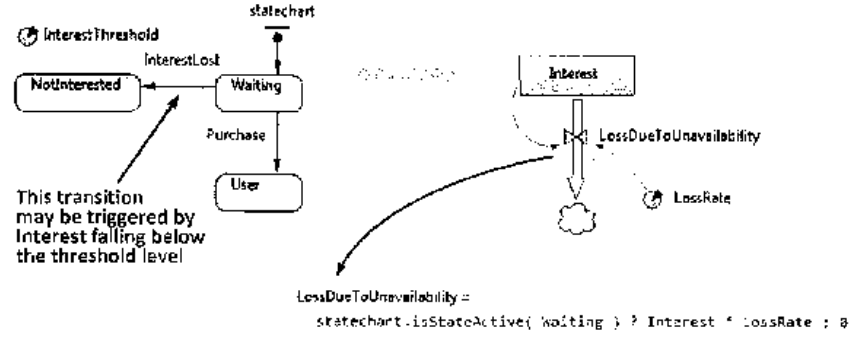
K. A message received by an agent modifies the internal SD stock. Such agent can be an element of a supply chain where orders and shipments are modeled as messages between agents. In the case K, the incoming message is of the type **Integer** (same as **int**) and is treated as the amount of raw material shipped to this agent. The internal dynamics of the agent are modeled in SD way, and the stock **RawMaterialInventory** is incremented upon the message arrival (see the **On message received** code of the agent). In more complex models the message can include the supplier ID, the type of product, and so on.

L. The bi-directional flow between the SD stocks is implemented outside the SD. The two stocks model available and occupied houses in the two neighborhoods of NYC. Both stocks are arrays with the dimension {**Midtwon, Soho**}. People (agents) may move from Midtown to SoHo and vice versa. When leaving, they decrement the bucket of the **Occupied** stock and increment the bucket of the **Available** stock. Then they do the reverse operation on the pair of buckets belonging to their new location.

I Statistics on agent population impact an SD variable (efficient)



J The statechart state impacts the SD rate



K The incoming message modifies the SD stock

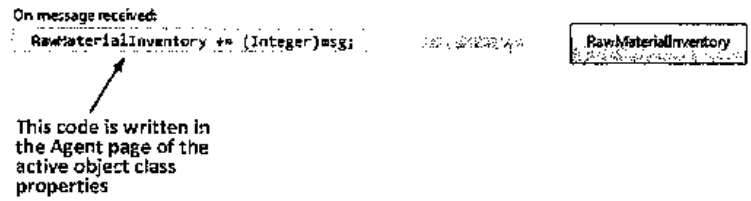

On message received:
RawMaterialInventory += (Integer)msg;
RawMaterialInventory
This code is written in the Agent page of the active object class properties

Figure 6.4 Discrete model elements impact SD (part 2)

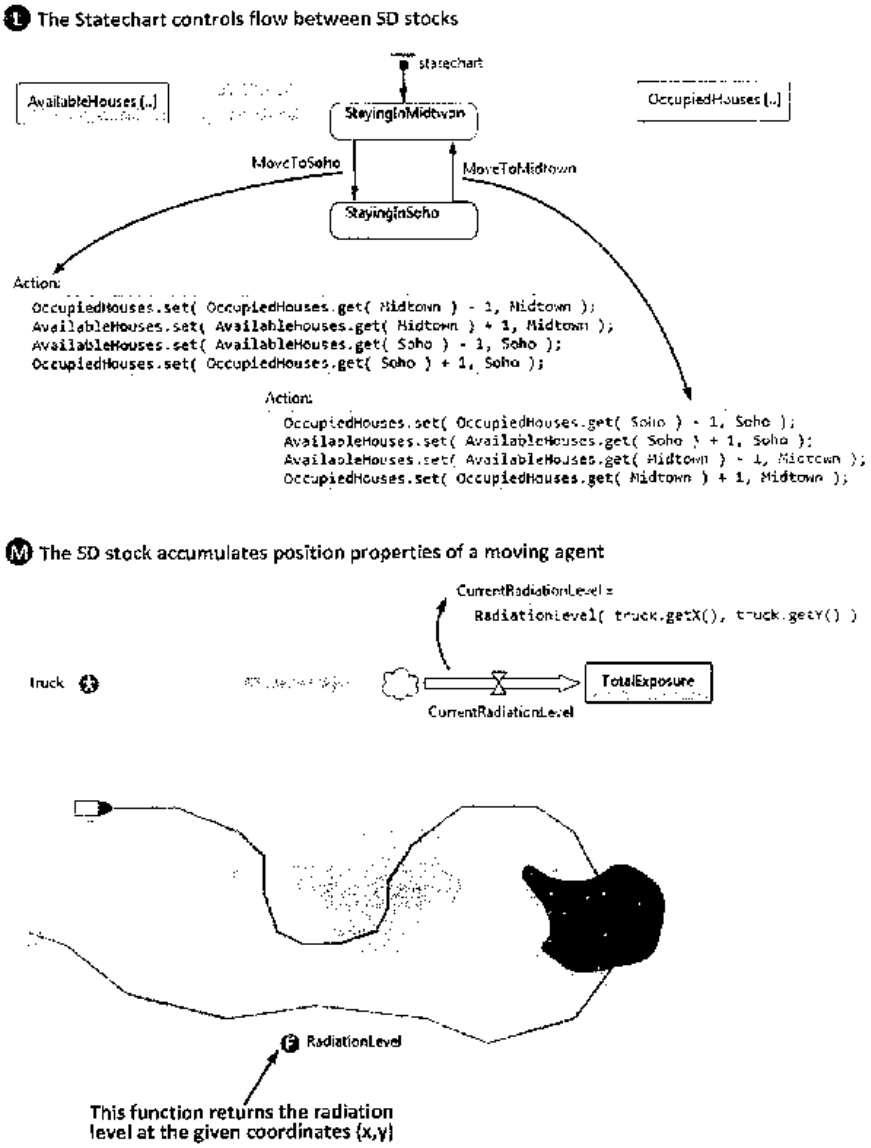

The Statechart controls flow between SD stocks
statechart
AvailableHouses [..]
StayingInMidtown
OccupiedHouses [..]
MoveToSoho
MoveToMidtown
StayingInSoho
Action:
OccupiedHouses.set(OccupiedHouses.get(Midtown) - 1, Midtown);
AvailableHouses.set(AvailableHouses.get(Midtown) + 1, Midtown);
AvailableHouses.set(AvailableHouses.get(Soho) - 1, Soho);
OccupiedHouses.set(OccupiedHouses.get(Soho) + 1, Soho);
Action:
OccupiedHouses.set(OccupiedHouses.get(Soho) - 1, Soho);
AvailableHouses.set(AvailableHouses.get(Soho) + 1, Soho);
AvailableHouses.set(AvailableHouses.get(Midtown) - 1, Midtown);
OccupiedHouses.set(OccupiedHouses.get(Midtown) + 1, Midtown);
The SD stock accumulates position properties of a moving agent
CurrentRadiationLevel =
RadiationLevel(truck.getX(), truck.getY())
Truck
TotalExposure
CurrentRadiationLevel
RadiationLevel
This function returns the radiation level at the given coordinates (x,y)

Figure 6.5 Discrete model elements impact SD (part 3)

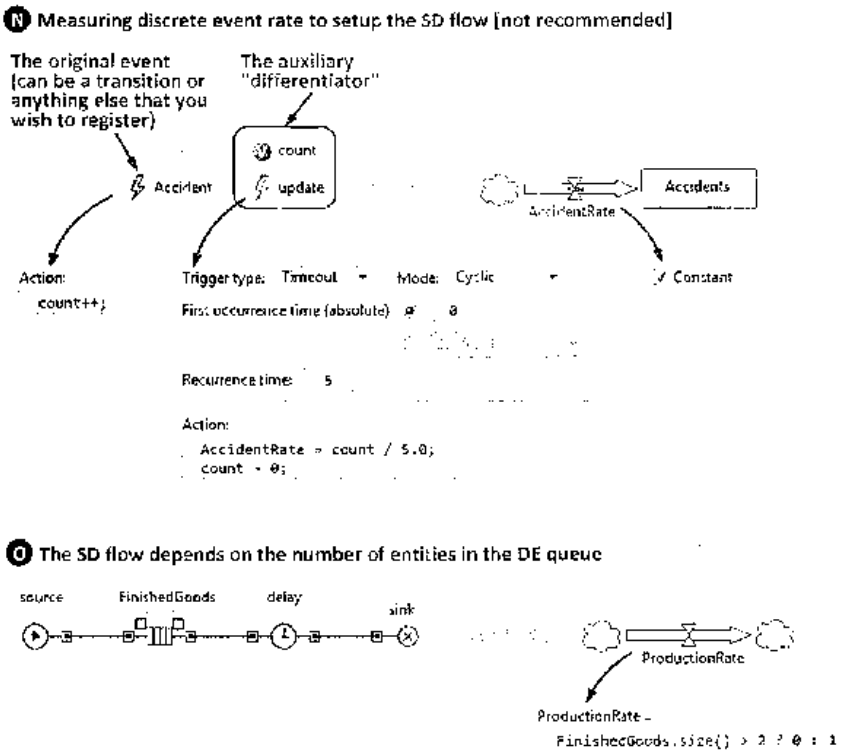


Figure 6.6 Discrete model elements impact SD (part 4)

M. The SD stock accumulates the "history" of the agent motion. This is an interesting example of SD-AB cooperation. The value of the stock **TotalExposure** is constantly updated as the mobile agent moves through the area contaminated by radiation. The value of the incoming flow **CurrentRadiationLevel** is set to the radiation level at the coordinates of the truck agent. As the truck moves or stays, the stock receives and accumulates a dose of radiation per time unit. Again, in the SD equation, we are referencing the agent and calling its function.

N. Measuring discrete event rate and feeding it to the SD flow. Suppose you want to create an SD reflection of the discrete event flow. The discrete events may model things like purchases, arrivals, accidents, decisions, and so on. The technique shown in Figure 6.5 M can be used, but it is not recommended. The technique works as follows: the auxiliary recurring event **update** counts the number of **Accident** occurrences within a fixed interval, and sets up the **AccidentRate** to the number

divided by the interval duration. This is, in fact, a rough derivative calculation. This implementation requires two auxiliary elements and the correct choice of the update interval.

Whenever possible, we recommend to directly update the stock, and not the flow. In this case, the original event Accident could increment the stock Accidents on each occurrence. This will be a 100% accurate solution without any unnatural constructs.

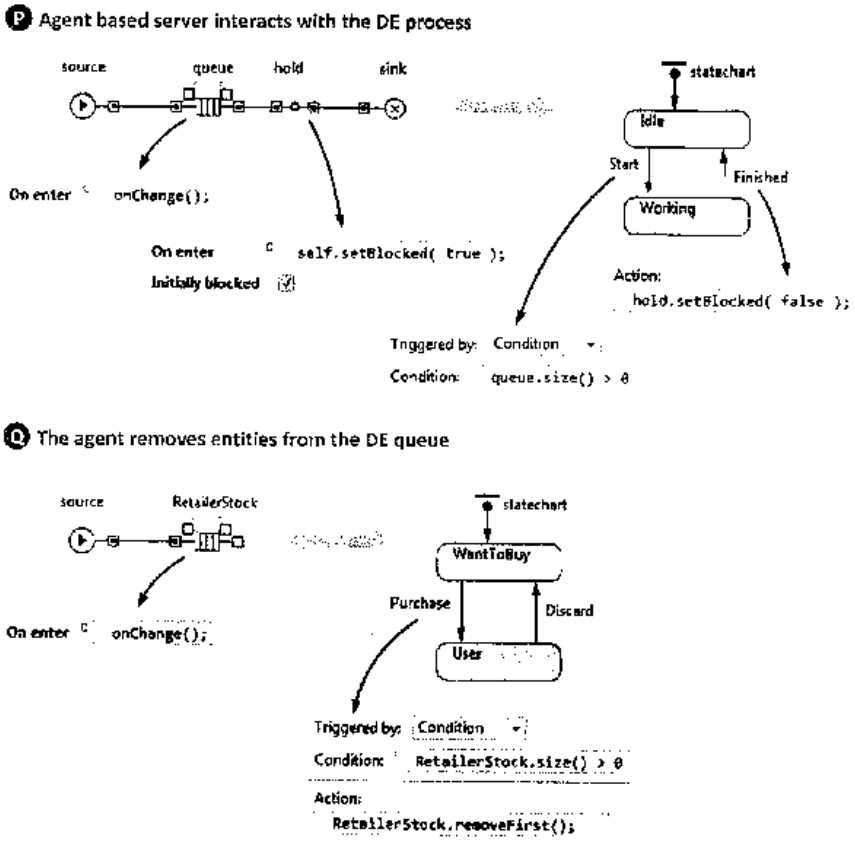


Figure 6.7 Agent based parts of the model interact with discrete event parts

0. Referencing DE objects in the SD formula. In this example, the flow ProductionRate switches between 0 and 1, depending on whether the finished products inventory (the number of entities in the queue FinishedGoods returned by

the function size()) is greater than 2 or not. Again, one can close the loop by letting the SD part control the production process.

Agent based <-> discrete event

P. A server in the DE process model is implemented as an agent. Imagine complex equipment, such as a robot or a system of bridge cranes. The behavior of such objects is often best modeled "in agent based terms" by using events and statecharts. If the equipment is a part of the manufacturing process being modeled, you need to build an interface between the process and the agent representing the equipment.

In this example, the statechart is a simplified equipment model. When the statechart comes to the state **Idle**, it checks if there are entities in the queue. If yes, it proceeds to the **Working** state and, when the work is finished, unblocks the **hold** object, letting the entity exit the queue. The **hold** object is set up to block itself again after the entity passes through.

The next entity will arrive when the equipment is in the **Idle** state. To notify the statechart, we call the function **onChange()** upon each entity arrival (see the **On enter** action of the queue).

Unlike in the models with continuously changing SD elements, in the models built of purely discrete elements events and transitions triggered by a condition, *do not monitor the condition continuously*. The event's condition is evaluated when the event is reset. The transition's condition is evaluated when the statechart comes into the transition's source state. And then the conditions are re-evaluated when something happens to the active object where they are located, or when its **onChange()** function is called.

Q. The agent removes entities from the DE queue. Here, the supply chain is modeled using discrete event constructs; in particular, its end element, the retailer stock, is a **Queue** object. The consumers are outside the discrete event part, and they are modeled as agents. Whenever a consumer comes to the state **WantToBuy**, it checks the **RetailerStock** and, if it is not empty, removes one product unit. Again, as this is a purely discrete model, we need to ensure that the consumers who are waiting for the product are notified about its arrival – that's why the code **onChange()** is placed in the **On enter** action of the **RetailerStock** queue.

In this simplified version, there is only one consumer whose statechart is located on the same canvas as the supply chain flowchart. In the full version there would be multiple agents-consumers and, instead of calling just **onChange()**, the retailer stock would notify *every consumer* in a loop, see the next example.

R. The incoming message injects entities into the DE process. In this example and next process flowcharts are put inside agents. This architecture can be used in supply chain models. In the case Figure 6.8 R the message received by the agent has a meaning of raw material shipment that starts the manufacturing process. In the **On message received** action the agent injects entities into the flowchart by calling the **Source** object function **inject()**. The source object **RawMaterialDeliveries** works in the "manual" mode when it does not generate entities unless **inject()** is called.

S. A process inside the agent initiates an outgoing message. Similarly, in Figure 6.8 S, the manufacturing process ends with the **Sink** object **ShipToConsumer**. When an entity representing the finished product unit gets there, before it disappears from the process, the **Sink** sends out the message using the agent interface function **send()**.

In this simplified model, the message is just an integer number 1, but in more realistic cases, you can send out the entity itself, or batches of multiple entities. On the receiving end, the entities can be directly injected into the process. The pair **Exit-Enter** should be used then, instead of the pair **Sink-Source**.

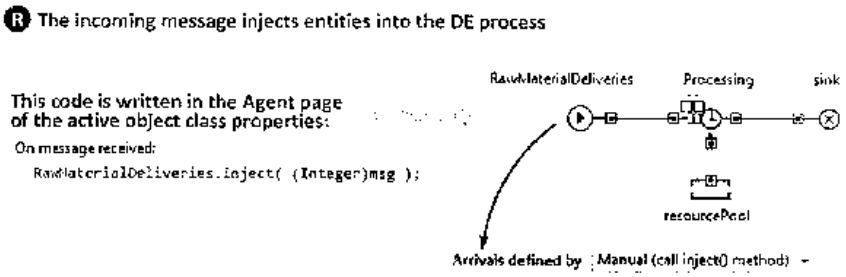

R The incoming message injects entities into the DE process
This code is written in the Agent page of the active object class properties:
On message received:
RawMaterialDeliveries.inject((Integer)msg);
RawMaterialDeliveries
Processing
sink
resourcePool
Arrivals defined by Manual (call inject() method)
S A process inside the agent initiates an outgoing message
RawMaterialDeliveries
Processing
ShipToConsumer
resourcePool
On enter send(1, consumer);

Figure 6.8 Incoming and outgoing messages interact with the DE process inside agent

T. The agent shares a 2D space with pedestrians and generates an event that affects them. Pedestrian flow is modeled using **AnyLogic Pedestrian Library**. The agent represents a terrorist who drives a car into a crowd and explodes. Everyone dies who happened to be closer than 50 meters to the car.

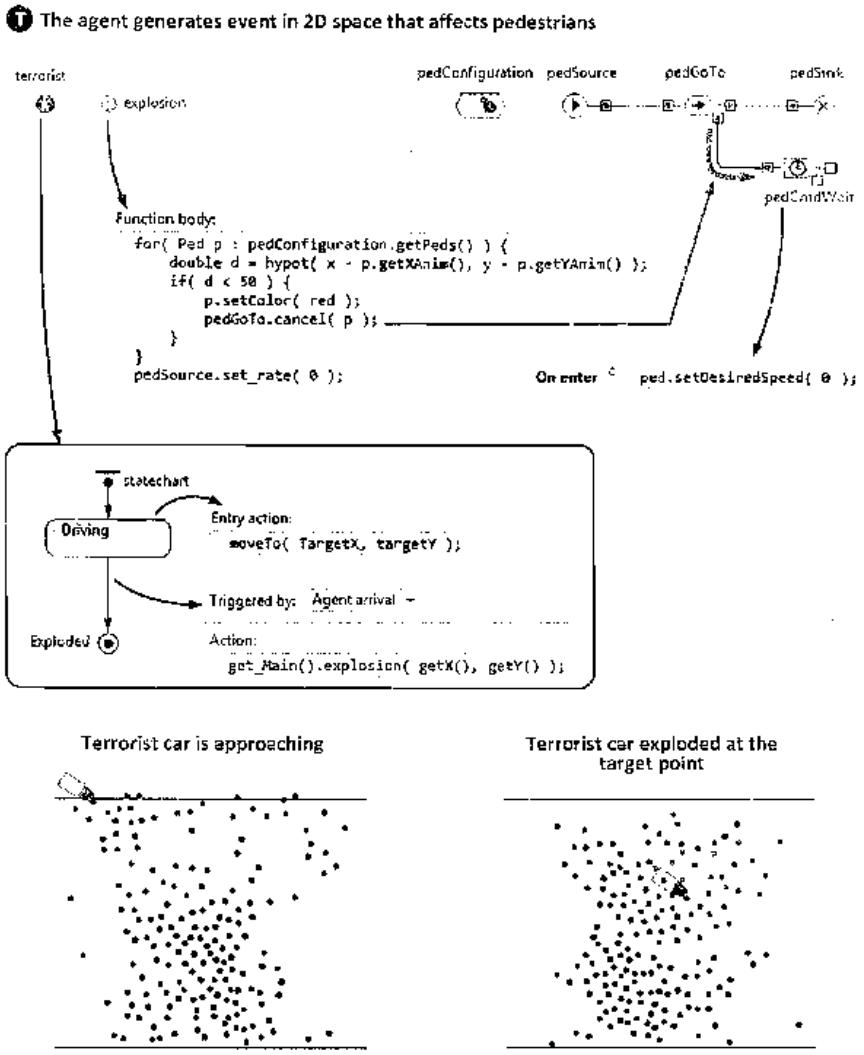


Figure 6.9 The agent generates event in 2D space that affects pedestrians

The interaction between the agent and the pedestrians is implemented in the function **explosion()**. The function accepts the arguments **x** and **y**, – the coordinates of the explosion. In the **for** loop, we iterate through all pedestrians (the collection returned by the function **getPeds()** of the **pedConfiguration** object). The pedestrian coordinates can be obtained via **getXAnim()** and **getYAnim()**. If the pedestrian is in the

death range, we extract him from the regular flow by calling the function **cancel()** of **pedGoTo**. The dead pedestrians exit **pedGoTo** via the "emergency" port and enter the **pedCmdWait** object where their speed is set to 0.

U. An entity in the process is at the same time an agent. Although this architecture is not present in Figure 6.10, it is often used to model process flows that can be changed or interrupted by events generated by entities' individual dynamics. Consider a patient who is in the middle of being treated in a hospital, and whose condition suddenly changes from normal to acute. This patient should be removed from the regular process and inserted into a special emergency treatment process.

U An Entity in DE process is at the same time an agent

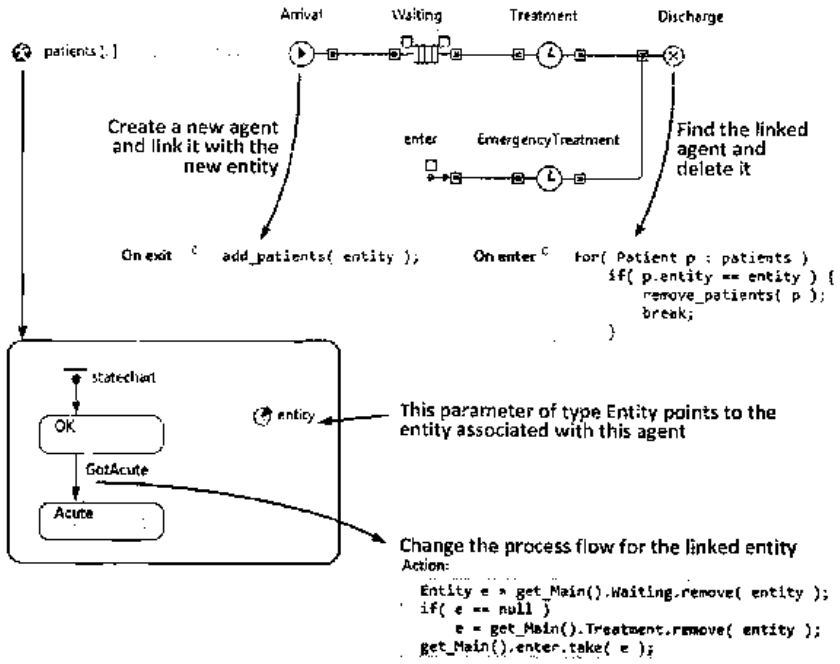


Figure 6.10 Each entity in the process has an agent associated with it

In the model shown in Figure 6.10, a population of agents (**patients**) is located on the same canvas as the process flowchart. The population is initially empty. As a new patient-entity is generated by the **Arrivals** source object, a new patient-agent is created and gets linked with the entity: the agent's parameter **entity** points to the newly created patient-entity. When the patient-entity is discharged, the **Discharge**

sink object searches for the patient-agent associated with the entity and deletes it from the model.

Each entity in the process, therefore, has an agent linked to it, and vice versa. The agent has individual dynamics: at any time, it can change its state from OK to Acute. When that transition happens, the agent looks for the linked entity, which can be either in the Waiting, or in the Treatment stage of the process. The agent removes the entity from the regular process and puts it into the enter object, so the patient-entity continues in the emergency treatment process.

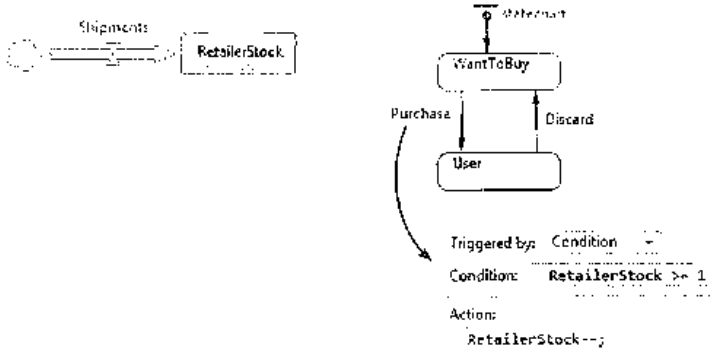
The interaction between the AB and DE parts can be further extended in this model. For example, the regular treatment performed on time can prevent the patient from entering the Acute state. On the technical side, one can implement the direct reverse link from the entity to the agent by adding the entity's field agent pointing to the associated agent. The procedure of agent deletion will then get a lot simpler.

Referencing model elements located in different active objects

For simplicity, in the previous examples, the statecharts, events, SD variables, and process flowcharts are often located on the same canvas (in the same active object). Inreal models, elements belonging to the different methods are often located in different active objects/agents and on different hierarchy levels. The only change in the model code would be the way the model elements reference one another: prefixes might be needed before their names. Figure 10.16 gives a general idea about the prefixes. Here, we will refactor example F to illustrate this.

The top case in Figure 6.11 is the original example, where the statechart modeling the consumer behavior is located on the same level as the system dynamics stock RetailerStock in the same Main active object. We want to have multiple consumers in the model; therefore we will place the statechart in the agent Consumer and place a population of consumers in the Main object, replacing the old statechart, see the bottom case in Figure 6.11. Because the RetailerStock is now located one level up from the statechart, we need to refactor the code in the Purchase transition to reflect this. The prefix "get_Main()." brings us to the direct container of a consumer – to the Main active object.

F The SD stock triggers a statechart transition, which, in turn, modifies the stock value



F [Refactored] Same, but now the statechart is inside the agent

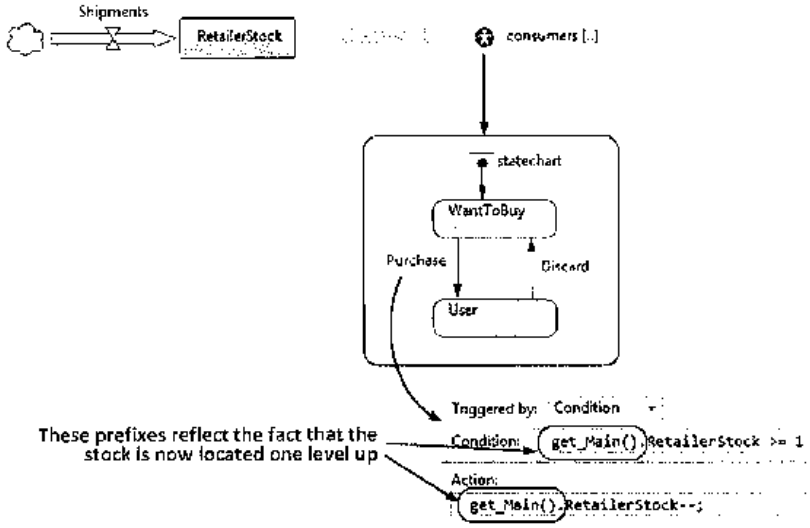


Figure 6.11 Discrete model elements impact SD (part 1)

The function **get_Main()** was automatically generated by AnyLogic into the **Consumer** active object, when it noticed you have embedded the **Consumer** into **Main**. If the **Consumer** had been embedded into, say, the **Market** object, the function **get_Market()** would have been generated instead. Should you have two populations of consumers, one in **MarketA**, and another in **MarketB**, two functions will be generated,

and the function **get_MarketB()** called in the consumer embedded into **MarketA** will return **null**.

The simulation performance of multi-method models

The AnyLogic simulation engine is specifically designed to efficiently execute a mixture of continuous dynamics with discrete events. It is also set up to support large numbers of parallel activities exhibited by agent based models. Two or three methods working together in the same model do not slow down the simulation or increase the memory footprint of the model. There are, however, some things you should keep in mind when designing multi-method models.

SD is typically simulated more slowly than other methods. The frequency of micro-steps of the numeric solver is typically higher than the frequency of the discrete events in the model. Moreover, the frequency of the numeric steps is constant, whereas the intervals between discrete events are irregular. In the case of a purely discrete model, the engine would just jump to the next event, no matter how far it is on the time axis. The numeric solver settings can be adjusted on the **Advanced** page of the experiment properties.

Each SD formula is evaluated multiple times at each numeric step. Therefore, the time complexity of formula evaluation affects a lot of the overall simulation performance. You should avoid including complex calculations (for example, loops traversing populations of agents or collections of entities) directly into the formulas. Consider the cases H and J in the example set previously considered.

If SD is present in the model, the conditions of events and transitions are also evaluated at each numeric step. This is done to ensure that the events and transition will be triggered exactly when their conditions turn true. The computational complexity of those conditions should also be kept low.

If you embed an SD stock and flow diagram into an agent, at runtime there will be as many independent copies of that diagram as there are agents. If, for example, the diagram contains 100 dynamic variables, in a population of 1,000 agents, there will be a system of 100,000 dynamic variables submitted to the numeric solver, and you should expect the corresponding impact on the simulation performance.

6.3. Examples

Example 6.22: Epidemic and clinic

We will create a simple agent based epidemic model and link it to a simple discrete event clinic model. When a patient discovers symptoms, he will ask for treatment in

the clinic, which has limited capacity. We will explore how the capacity of the clinic affects the disease dynamics. This model was suggested in 2012 by Scott Hebert, a consultant at AnyLogic North America.

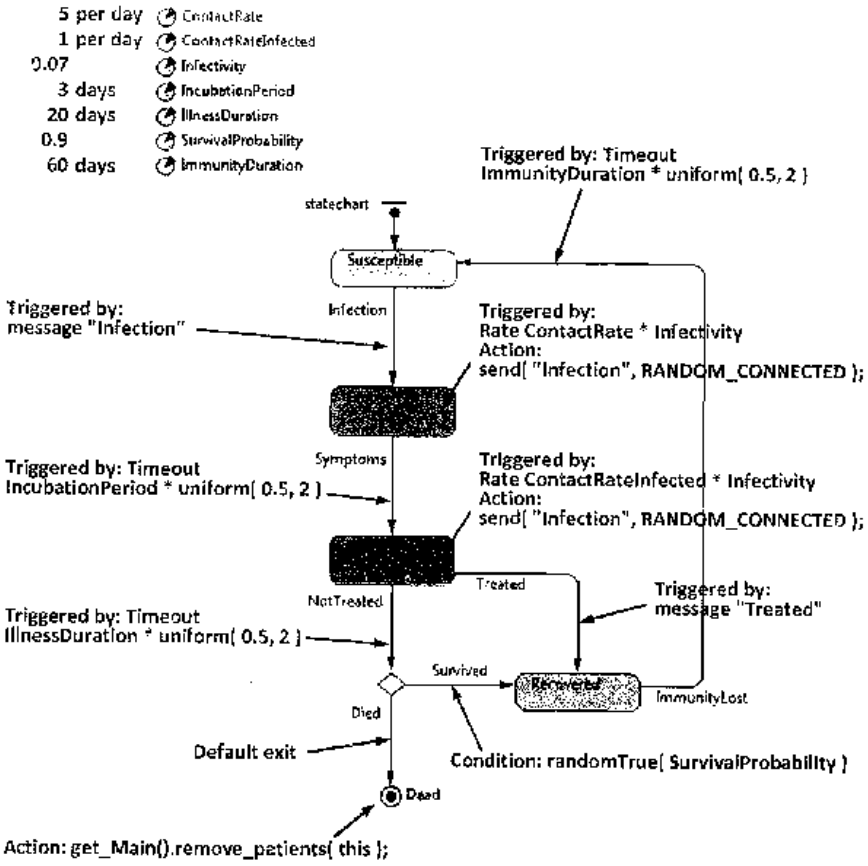


Figure 6.12 The patient behavior statechart

▶ **Create the agent population:**

1. Create a new model.
2. Drag the **Agent Population** from the **General** palette onto the graphical editor of the **Main** object.
   In the first page of the wizard, change the **Agent class name** to **Patient**, and the **Population name** to **patients**. Set the initial number of agents to 2000.

In the next page, of the wizard set the space **Width** to 650 and **Height** to 200.Set **Network** type to **Distance based** with range 30. Click **Finish**.

3. Run the model. The agents are randomly distributed in the rectangular space and, so far, no activity is going on.

We have created a population of 2,000 agents. There is a network in the population: if the distance between two agents is less than 30, they are linked. The network and layout settings can be changed in the settings of the **environment** object. The next step is to define the behavior of our patient. We will use a statechart (see Chapter 7) for that.

The statechart is similar to the classical SEIR statechart ("Compartmental models in epidemiology", n.d.). The patient is initially in the **Susceptible** state, where he can be infected. Disease transmission is modeled by the message **"Infection"** sent from one patient to another. Having received such a message, the patient transitions to the state **Exposed**, where he is already infectious, but does not have symptoms. After a random incubation period, the patient discovers symptoms and proceeds to the **Infected** state. We distinguish between the **Exposed** and **Infected** states because the contact behavior of the patient is different before and after the patient discovers symptoms: the contact rate in the **Infected** state is 1 per day, as opposed to 5 in the **Exposed** state. The internal transitions (see Section 7.3) in both states model contacts. We model only those contacts that result in disease transmission; therefore, we multiply the base contact rate by **Infectivity**, which, in our case, is 7%.

There are two possible exits from the Infected state. The patient can be treated in a clinic (and then, he is guaranteed to recover), or the illness may progress naturally without intervention. In the latter case, the patient can still recover with a high probability (90%), or die. If the patient dies, it deletes himself from the model, see the **Action** of the **Dead** state. The completion of treatment is modeled by the message **"Treated"** sent to the agent. So far, this message is never received, because we have not created the clinic model yet.

The recovered patient acquired a temporary immunity to the disease. We reflect this in the model by having the state **Recovered**, where the patient does not react to the message **"Infection"** that may possibly arrive. At the end of the immunity period the transition **ImmunityLost** takes the patient back to the **Susceptible** state.

Note that as long as we have defined the parameters *inside the agent*, we can make their values different for different agents. In our model, however, for simplicity, they are the same throughout the whole population.

▶ **Define the patient behavior:**

4. Open the editor of the **Patient** object.

5. Add seven parameters with the default values shown at the top left of Figure 6.12.
6. Draw the statechart as shown in Figure 6.12.
7. In the **Entry action** field of each state, type the code that changes the color of the patient animation into the color of the state, for example, in the **Entry action** of the state **Exposed** type: `person.setFillColor( darkOrange );` (this is not shown in Figure 6.12).
8. Open the properties of the **Main** object and type the following code in the **Startup code** field:
   ```
   for( int i=0; i<5; i++ )
     patients.random().receive( "Infection" );
   ```
   This will infect five randomly chosen people in the beginning of the simulation.
9. Run the model for a while.

The contagious disease spreads around the initially infected agents (remember that our network of contacts is based on distance). The epidemic does not end after the first wave, because the immunity period is not long enough. We will now add a chart to view the type of the system dynamics.

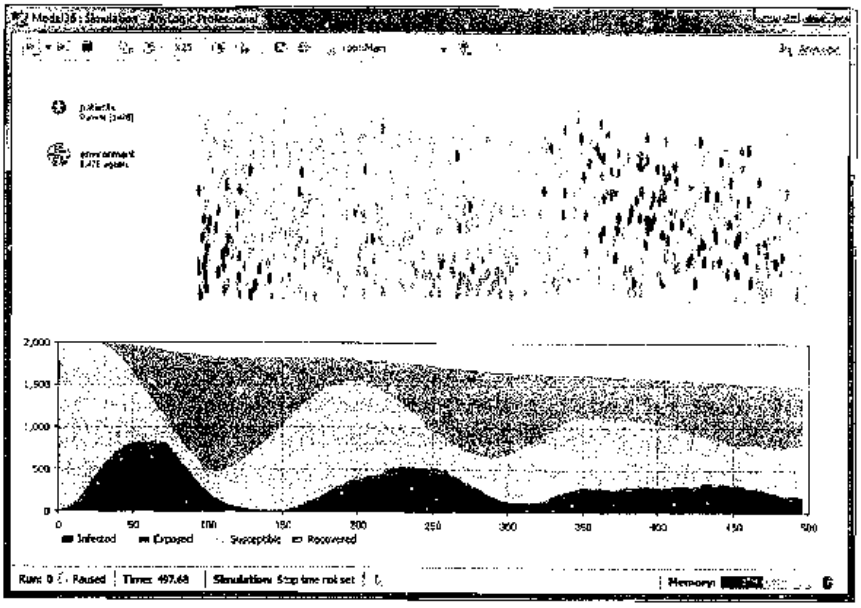


Figure 6.13 Oscillation in the epidemic model

▶ **Collect statistics and display graphs:**

10. Open the editor of **Main**, select the **patients** population and open the **Statistics** page of its properties. Add statistics with the name **NSusceptible** of type **count** and with the condition **item.statechart.isStateActive(item.Susceptible)**. This will generate the function **NSusceptible()** of the population that will count and return the number of patients who are in the **Susceptible** state.
11. Add three other statistics for the states **Exposed**, **Infected**, and **Recovered**, respectively.
12. Drag the **Time stack chart** from the **Analysis** palette onto the **Main** object, and place it below the area occupied by the agents. Set the **Time window** of the chart to 500, and **Display up to** 500 latest samples.
13. Add four data items to the chart corresponding to the four statistics functions you have created. The first item, for example, has the title Infected, **orangeRed** color, and value **patients.NInfected()**.
14. Select the **Simulation** experiment in the **Projects** tree, and open the **Presentation** page of its properties. Set **Execution mode** to **Real time with scale** 25. This will make things happen faster.
15. Run the model. Now you can see oscillation and the gradual decrease of the total population due to deaths, see Figure 6.13.

The next step is to add the clinic, and let the patients be treated there. Our clinic will be modeled via a very simple discrete event model: **Queue** for the patients waiting to be treated and **Delay** modeling the actual treatment. However, unlike in pure discrete event models, the entities in this process will not be generated by the **Source** object, but injected by the agents. The communication scheme between the patients-agents and the clinic process is shown in Figure 6.14.

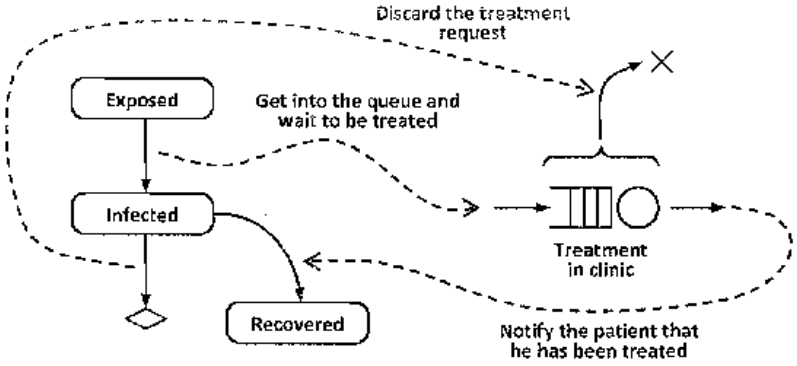


Figure 6.14 Interface between the agent based and the discrete event parts of the model

Once the patient discovers symptoms, he will create an entity – let's call it "treatment request" – and inject the entity into the clinic process. Once the treatment is completed, the entity will notify the patient by sending him a message "Treated" that will cause the patient to transition to the Recovered state. If, however, the patent is cured or dies before the treatment is completed, he will discard his treatment request by removing it from whatever stage in the process it is. On the technical side, we need:

- The entity that will carry the reference to the patient
- The ability to remove the entity originated by a particular patient from the process.

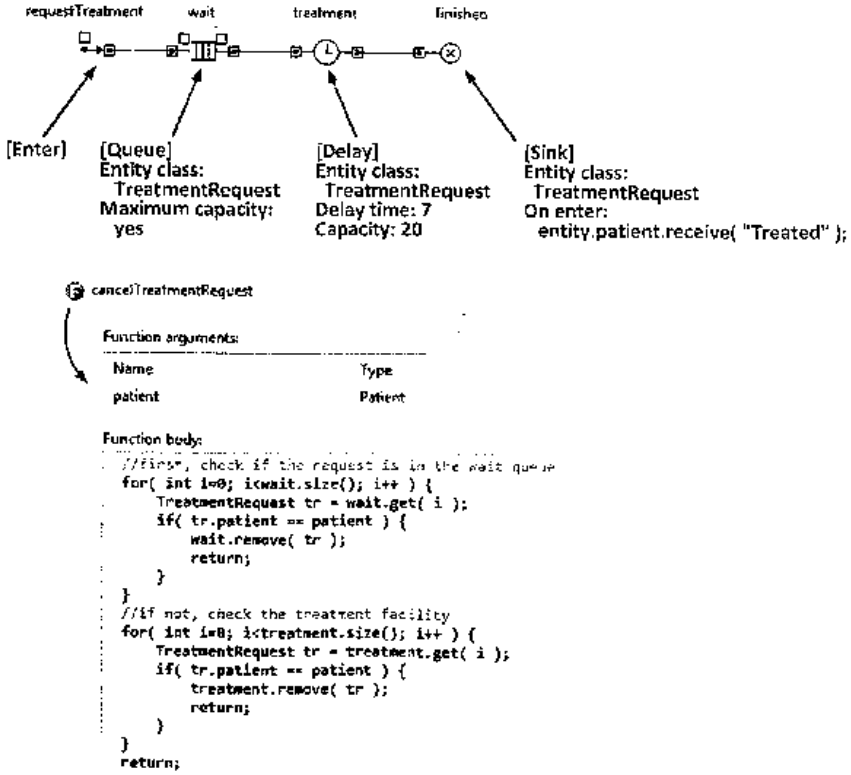


Figure 6.15 The discrete event part – the model of clinic

▶ **Create the new entity type TreatmentRequest:**

16. Right-click the model item in the **Projects** tree, and select **New | Java class** from the context menu. In the wizard, name the class **TreatmentRequest** and set its **Superclass** to **Entity**. Click **Next**.
17. On the next wizard page add, one field with the name **patient** and type **Patient**. Click **Finish**, and close the Java editor that opens – you will not need it.
18. Open the editor of **Main** and put together the process flowchart as shown in Figure 6.15. Use objects from the **Enterprise library** palette.
19. Remaining in **Main**, create the function **cancelTreatmentRequest()** with the parameters and the body code as shown in Figure 6.15.

We created a custom entity class **TreatmentRequest** that has a field **patient** – this will be the reference to the patient who originated the treatment request. In the process flowchart, we identified that entities passing through the **wait**, **treatment**, and **finished** objects are not of the generic **Entity** class, but of its subclass **TreatmentRequest**. This is necessary because we plan to use the **patient** field of those entities. For example, when the treatment is finished, the finished object sends a message "Treated" to the patient referenced by the entity before disposing of the entity.

We also specified that the queue has infinite capacity, that the treatment takes exactly seven days, and that there are only 20 beds in the clinic, so only 20 patients can be treated simultaneously.

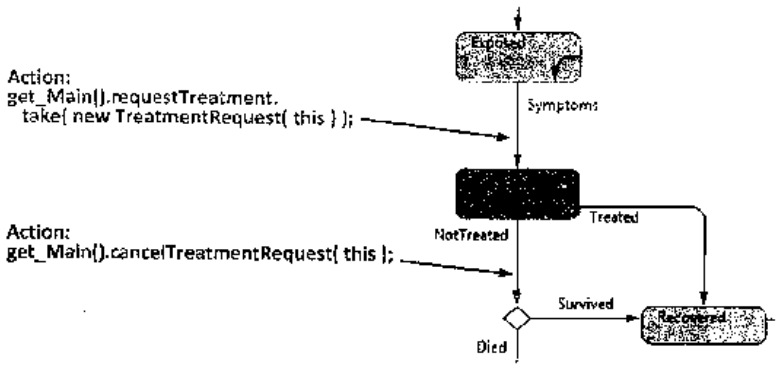


Figure 6.16 The patient behavior modified to communicate with the clinic process

Finally, we prepared the function **cancelTreatmentRequest()** that will be called by patients who got well on their own or died before getting chance to be treated. That function uses the API of the **Queue** and **Delay** objects to search for a particular entity in them and remove it.

The remaining task is to modify the behavior of Patient to link it to the model of clinic. Remember that since the clinic process is located one level above the patient's statechart, in the Main object, the clinic objects and functions should be preceded by the prefix get_Main(). The Java word "this" references the object to which the code belongs, in this case, the patient.

▶ **Incorporate treatment in the clinic into the patient behavior:**

20. Add the actions to the statechart transitions Symptoms and NotTreated as shown in Figure 6.16.
21. The model is complete. Run the model.

Now, the model shows a different dynamic, or, to be more precise, a different range of dynamics. The oscillations are still possible, but a possibility also exists that the epidemic will end after the first wave, as shown in Figure 6.17. You may experiment with different clinic capacities to figure out the number of beds needed in order to treat everybody on time and prevent the further waves of the epidemic.

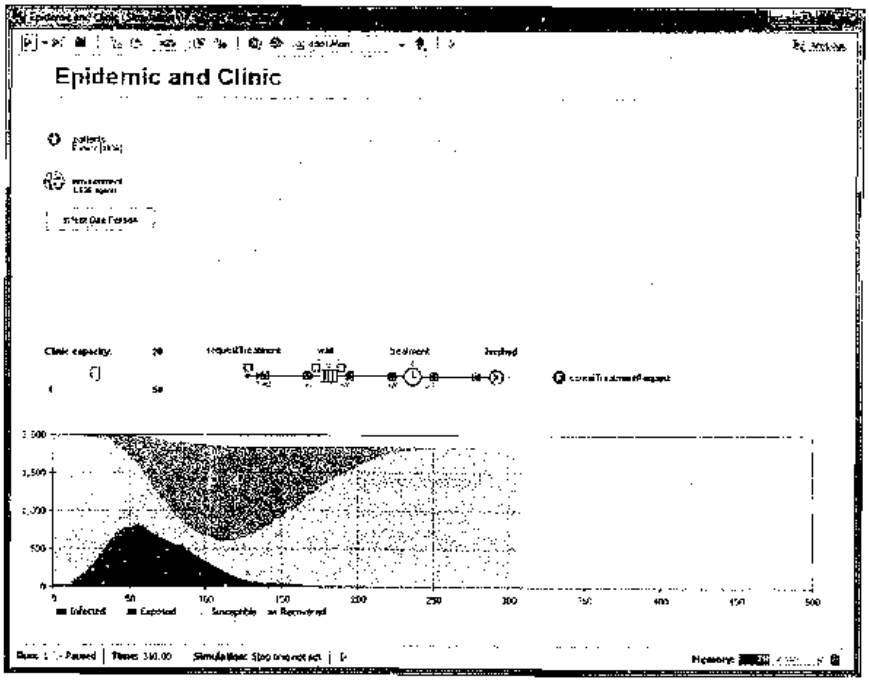


Figure 6.17 The behavior of the integrated epidemic and clinic model

Example 6.23: Consumer market and supply chain

We will model the supply chain and sales of a new product in a consumer market in the absence of competition. The supply chain will include the delivery of the raw product to the production facility, production, and the stock of the finished products. The QR inventory policy will be used. Consumers are initially unaware of the product; advertizing and word of mouth will drive the purchase decisions. The product has a limited lifetime, and 100% of users will be willing to buy a new product to replace the old one.

We will use discrete event methodology to model the supply chain, and system dynamics methodology, namely, a slightly modified Bass diffusion model, to model the market. We will link the two models through the sales events.

▶ **Create the supply chain model:**

1. Create a new model, and put together the process flowchart as shown in Figure 6.18. All parameters not mentioned there should be left at their default values.
2. Open the properties of the **Main** object, and type this code in the **Startup code** field: **Supply.inject(OrderQuantity);**. This will load the supply chain with some initial product quantity.
3. Run the model.

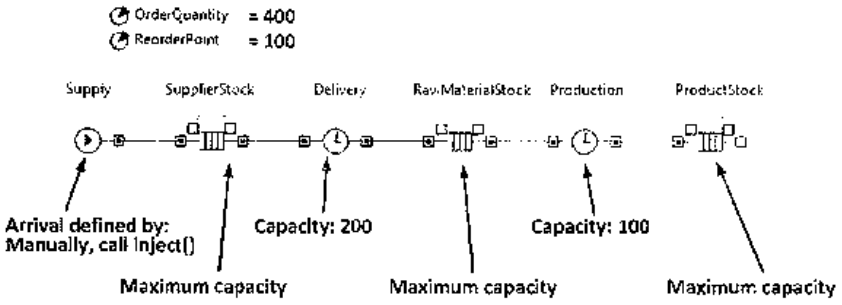


Figure 6.18 The discrete event model of the supply chain

Four hundred items of the product are produced and accumulate in the **ProductStock**. Nothing else happens in the model (the **Supply** object is set up to not generate any new entities, unless explicitly asked to do so). The inventory policy is not yet present in our model.

▶ **Create the market model:**

4. Create the stock and flow diagram as shown in Figure 6.19. You can place the diagram below the supply chain flowchart.

5. Run the model.

The supply chain still produces the 400 items and stops, and the potential clients gradually make their purchase decisions, building up the **Demand** stock.

The difference of our market model from the classical Bass diffusion model with discards (Sterman, 2000) is that the users, or adopters, stock of the classical model is split into two: the **Demand** stock and the actual **Users** stock. The adoption rate in this model is called **PurchaseDecisions**. It brings **PotentialUsers** not directly into the **Users** stock, but into the intermediate stock **Demand**, where they wait for the product to be available. The actual event of sale, i.e., "meeting" of the product and the customer who wants to buy it, will be modeled outside the system dynamics paradigm.

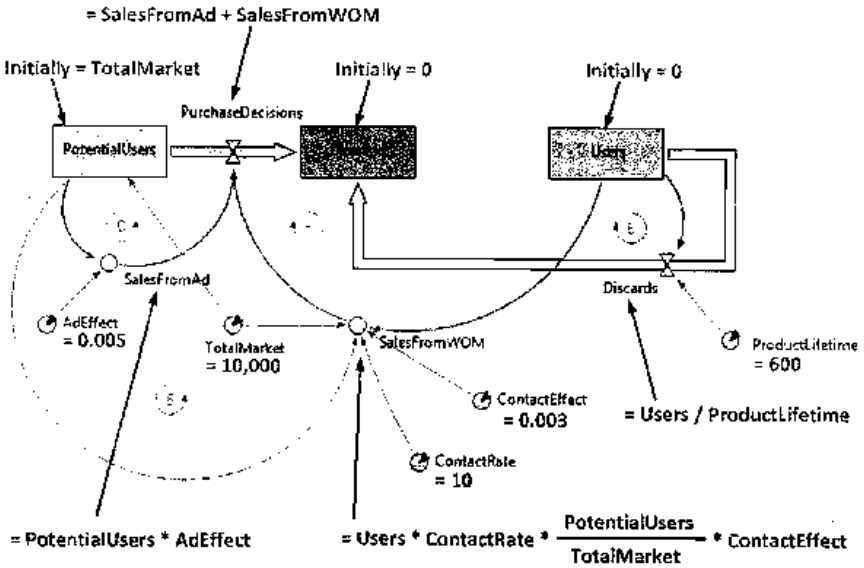


Figure 6.19 The system dynamics model of the market (the modified Bass model)

Before linking the two models we will put a couple of charts on the model UI so we can better view the dynamics.

▶ **Create the charts to view the supply chain and the market dynamics:**

6. Add the time stack chart and the time plot of the data items as shown in Figure 6.20. Set the **Time window** of both charts to 200 and let the charts **Display up to** 200 items collected every time unit.
7. Run the model again.

Note that in the current version of the model, the only reason for the potential users to make a purchase decision is advertizing. The word of mouth effect is not yet working because nobody has actually purchased a single product item.

How do we link the supply chain and the market? We want to achieve the following:

- If there is at least one product item in stock and there is at least one client who wants to buy it, the product item should be removed from the **ProductStock** queue, the value of **Demand** should be decremented, and the value of **Users** should be incremented, see Figure 6.21 A.

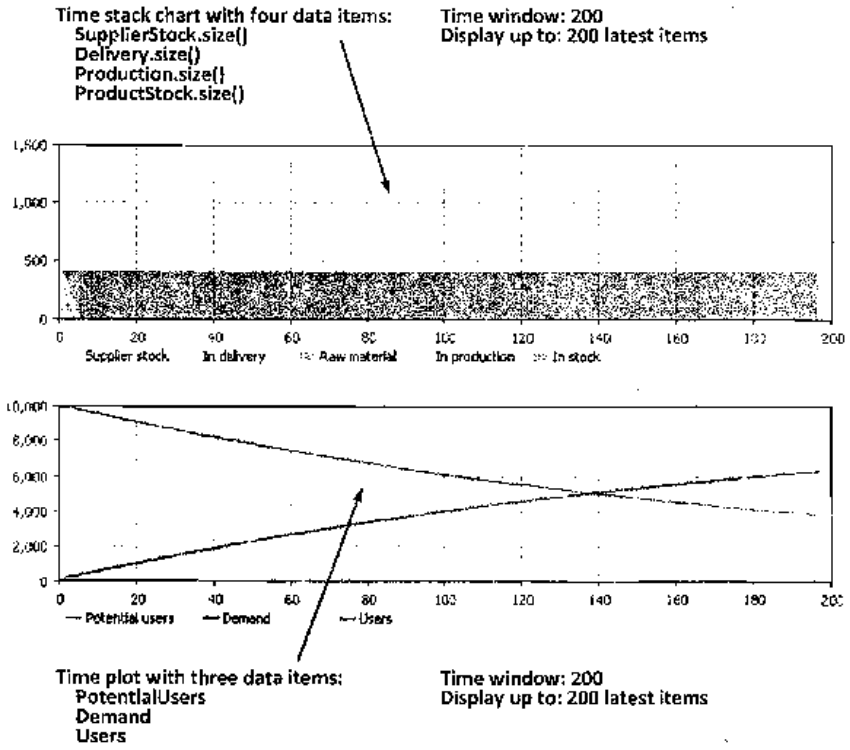


Figure 6.20 The charts in the supply chain and market model

We, therefore, have a condition and an action that should be executed when the condition is true. The AnyLogic construct that does exactly that is the condition-triggered event (see Section 8.2).

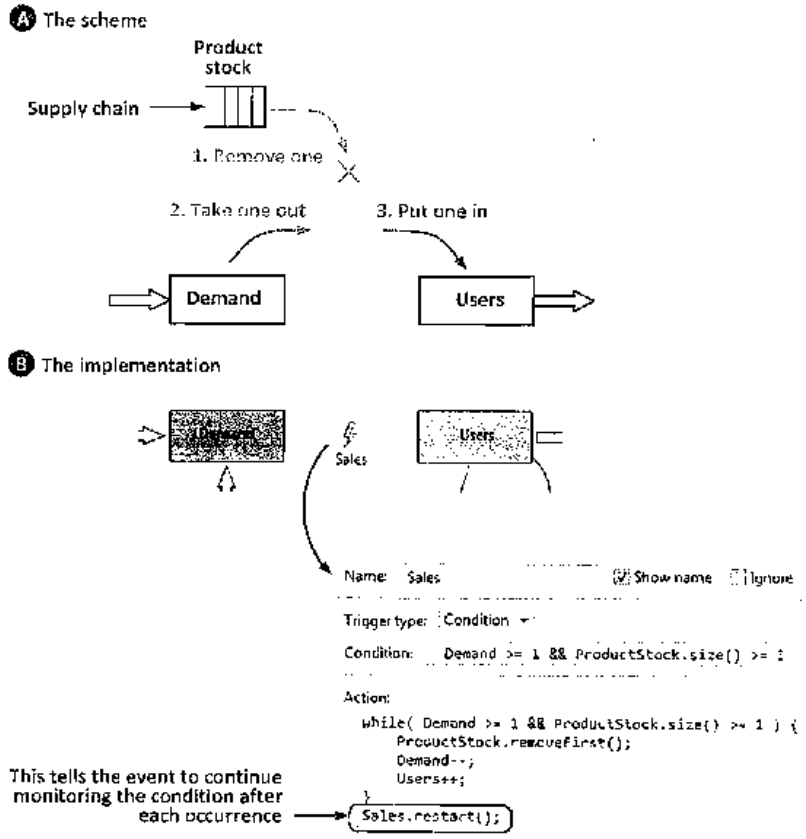


Figure 6.21 Linking the supply chain and the market: the scheme and the implementation

▶ **Link the supply chain and the market models with a condition event:**

8. Drag the **Event** object from the **General** palette, and place it in between the **Demand** and **Users** stocks (the location of the event object does not matter, but this way we can show that the event implements the missing Sales rate). Change the event name to **Sales**.
9. Specify the trigger and the action of the event, as shown in Figure 6.21 B.
10. Run the model.

In the presence of continuous dynamics in the model the condition of the event is evaluated at each numeric micro-step. Once the condition evaluates to true, the event's action is executed.

We put the sale into a **while** loop, because a possibility exists that two or more product items may become available simultaneously, or the **Demand** stock may grow by more than one unit per numeric step. Therefore, more than one sale can potentially be executed per event occurrence.

By default, the condition event disables itself after execution. As we want it to continue monitoring the condition, we explicitly call **restart()** at the end of the event's action.

The sales start to happen. The 400 items produced in the beginning of the simulation disappear in about a week. The **Users** stock increases up to almost 400; it then slowly starts to decrease according to our limited lifetime assumption. And, since we have not implemented our inventory policy yet, no new items are produced. This is the last missing piece of the model. We will include the inventory policy in the same **Sales** event; the inventory level will be checked after each sale.

▶ **Implement the QR inventory policy:**

11. Modify the Action code of the Sales event as shown below:

```
while( Demand >= 1 && ProductStock.size() >= 1 ) {
  //execute sale
  ProductStock.removeFirst(); //remoove a product unit from the stock (DE)
  Demand--; //remove a waiting client from Demand stock (SD)
  Users++; //add a (happy) client to the users stock (SD)
}
//apply inventory policy
int inventory = //calculate inventory
    ProductStock.size() + //in stock
    Production.size() + //in production
    RawMaterialStock.size() + //raw product inventory
    Delivery.size() + //raw product being delievered
    SupplierStock.size(); //supplier's stock
if( inventory < ReorderPoint ) //QR policy
  Supply.inject( OrderQuantity );
//continue monitoring
Sales.restart();
```

12. Run the model.

Now, the supply chain starts to work as planned, see Figure 6.22. (Here the inventory chart and the market charts are combined: the one was dragged on top of the other, and the labels were put on different sides.)

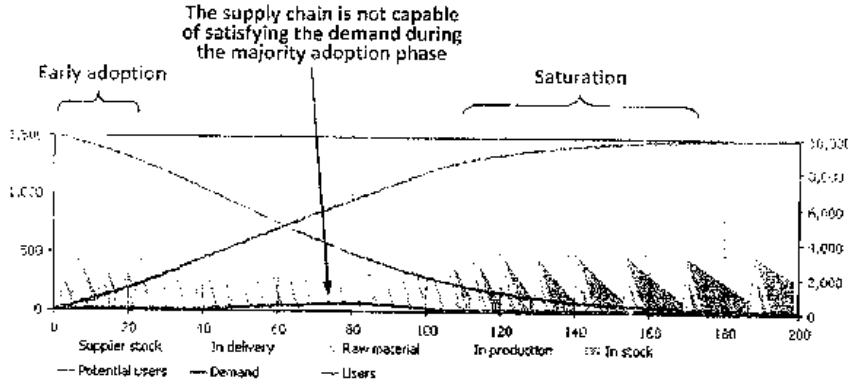


Figure 6.22 The supply chain dynamics pattern changes as the market gets saturated

During the early adoption phase the supply chain performs adequately, but as the majority of the market starts buying, the supply chain cannot keep up with the market. In the middle of the new product adoption (days 40-100), even though the supply chain works at its maximum throughput, still the number of waiting clients remains high. As the market becomes saturated, the sales rate reduces to the replacement purchases rate, which equals the **Discard** rate in the completely saturated market, namely **TotalMarket / ProductLifetime** = 16.7 sales per day. The supply chain handles that easily.

? Make the supply chain adaptive. Try to minimize the order backlog and at the same time minimize the inventory by adding feedback from the market model to the supply chain model.

Example 6.24: Product portfolio and investment policy

A company develops and sells consumer products with fairly short lifecycle. After the product has been successfully launched, its revenue peaks, and then falls, as shown in Figure 6.23. To keep the business going the company has to maintain a continuous process of new product research and development. Part of the company revenue is therefore reinvested in R&D, and another significant part is spent on introducing new products. We will investigate how the investment policy affects the business.

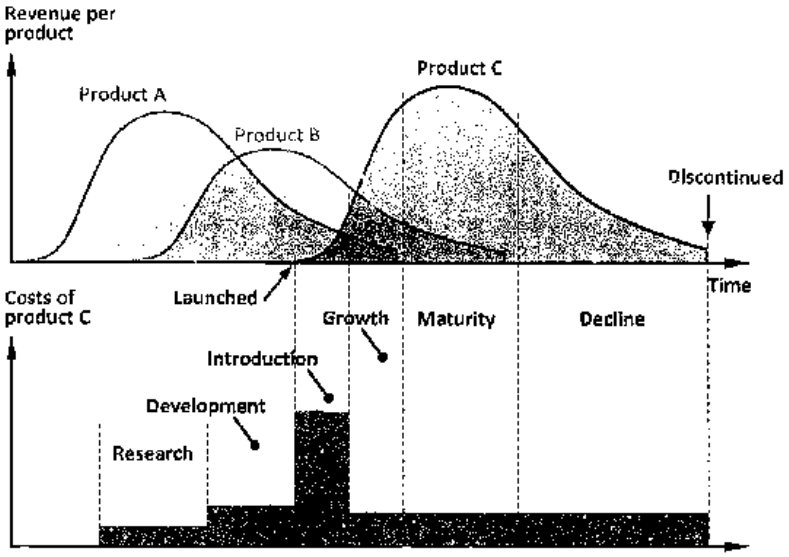


Figure 6.23 The product lifecycle and associated costs

We will make some simplifying assumptions:

The product lifecycle:

- The duration of the research phase of a product is uniformly distributed between 1.5 and 6 years. During this phase, each product is assigned a random "success factor", which is uniformly distributed between 0 and 1. At the end of the research phase, the product is killed if its success factor is less than 0.5; otherwise, it proceeds to the development phase.
- The duration of the development phase is also uniformly distributed between 1 and 3 years. During the development phase, the initial success factor is modified by adding a random number uniformly distributed between -0.3 and 0.3. At the end of the development phase, the project is killed if its success factor is less than 0.5; otherwise, it is released to the market.
- When the product is launched, its success factor is once again modified by adding a random number between -0.3 and 0.3. This value of the success factor then stays the same. As you can see, the value is between 0.2 and 1.6.

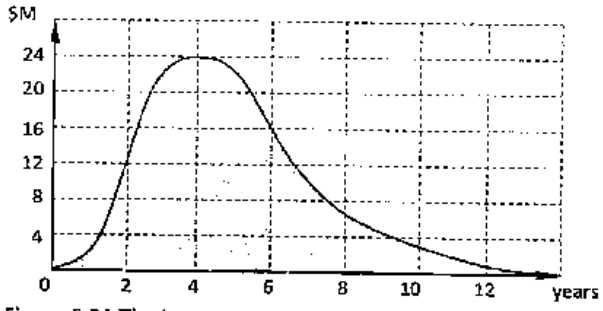


Figure 6.24 The base revenue curve of a product

Revenue and cost:

- While in market, all products have the same base revenue over time curve (see Figure 6.24), and the actual revenue equals base times the "success factor".
- The first two years of the product in market are considered as "introduction period". Any time after the introduction period the product is discontinued if its annual revenue falls below $5M.
- The annual cost of research is $0.5M per product and is same for all products. The annual cost of development is $1M per year. The cost of introducing a new product is $10M, which is spent evenly during the two years. In addition there is a one-time fixed cost of $0.5M for starting a new R&D project.
- After the introduction period we will assume no cost per product in market, in other words, we will treat the revenue as revenue after production and distribution costs.

Investment policy:

- A fraction of the company revenue goes into "investment capital". All R&D costs are paid from the investment capital.
- The company has a limited R&D capacity and cannot perform more than 100 projects concurrently.
- Once the company determines that the accumulated investment capital is greater than (the number of ongoing R&D projects + 1) times "average project cost" ($3M is assumed), a new project is started.
- The invested fraction of the revenue is determined as follows. If the accumulated investment capital is greater than the R&D capacity times "average project cost", no money is invested. Otherwise, 20% of the revenue goes into the investment capital stock.

- The remaining part of the revenue goes into the "main capital" stock, and product introduction costs are paid from there.

Assumptions, as you can see, are quite strong. For example, R&D projects may be killed at only two points: at the end of the research phase and at the end of the development phase, but not halfway through. The money spent on introducing the new product does not vary from product to product, the product lifecycles are similar, and there are no complete market failures and no great, long-lasting successes. All these things can be incorporated into the simulation model, but for the purpose of demonstrating the interaction of different modeling methods a simpler model will work just fine. Of course, all numeric values previously given are not fixed and will become the model parameters.

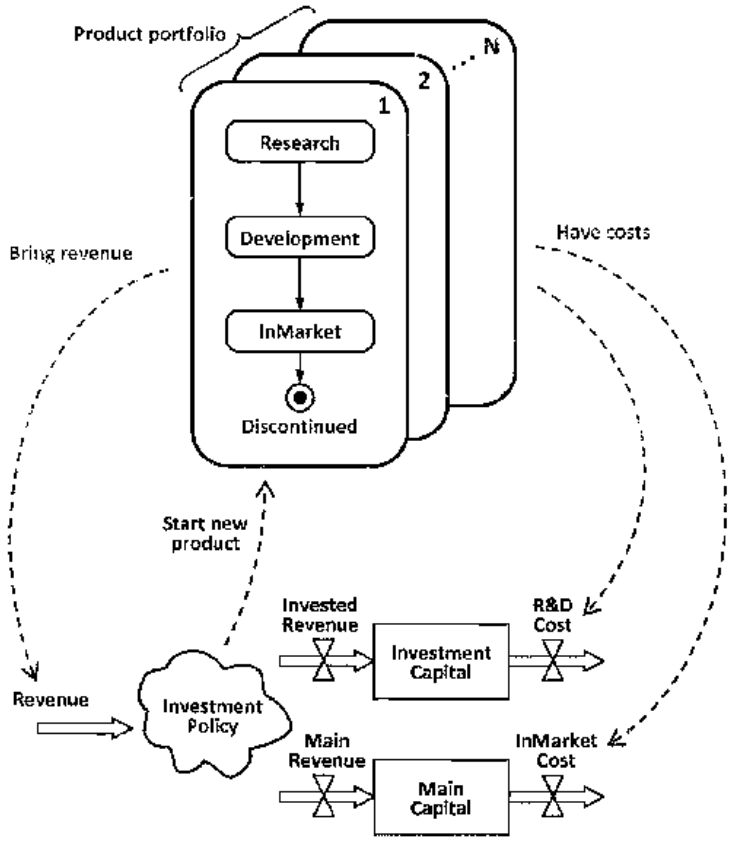


Figure 6.25 Architecture of the product portfolio and investment policy model

We will model each product individually as an agent (as you know, agents in agent based model are not necessarily people – they can be anything), so the product portfolio will be a population of agents. The company finances and investment policy will be a system dynamics component. The overall architecture of the model is shown in Figure 6.25. The product lifecycle is naturally represented as a statechart that starts in the Research state and is deleted from the model when it is discontinued or killed. The implementation of the interface between the products-agents and the system dynamics part will be clear from the step by step description that follows.

▶ **Create the Product agent and the portfolio population:**

1. Create a new model. Drag the **Agent Population** to **Main**, and specify the following parameters: **Agent class name**: **Product**, **Population name**: **portfolio**, **Initial population size**: 100, **Environment**: **Do not use.**
2. Open the editor of **Product**. Delete the default animation shape, and add the **Oval** from the **Presentation** palette instead. Call it **bubble**. We will animate the product portfolio in the form of a bubble chart.
3. Open the **Agent** page of the **Product** properties. Uncheck the checkbox **Environment defines initial location**, and set the **X** initial coordinate to **uniform(600)** and **Y** to **uniform(-60)**. This will distribute the bubbles randomly across a certain area.
4. In the same property page, set the **Velocity** of the agent to **100**. As the product will progress through the lifecycle phases, its bubble will move.
5. Open the editor of **Main**, and move the animation of the product portfolio (the bubble) to, say, (200,350). We need some space above, as the bubbles will move up.
6. Return to the editor of **Product**, and create parameters , variables, and the statechart as shown in Figure 6.26. As you can see, the numeric values in the problem statement have become parameters in the model.

For example, the duration of the development phase was originally specified as uniformly distributed between 1 and 3 years. In the model, the parameter **DevelopmentTime** has initial value of 2, and in the timeout expression in the **DevelopmentCompleted** transition, it is multiplied by a random coefficient taking values between 0.5 and 1.5, which gives us the distribution we need. Should we decide to change the average value of the development duration, the resulting interval will correctly follow it.

7. Run the model.

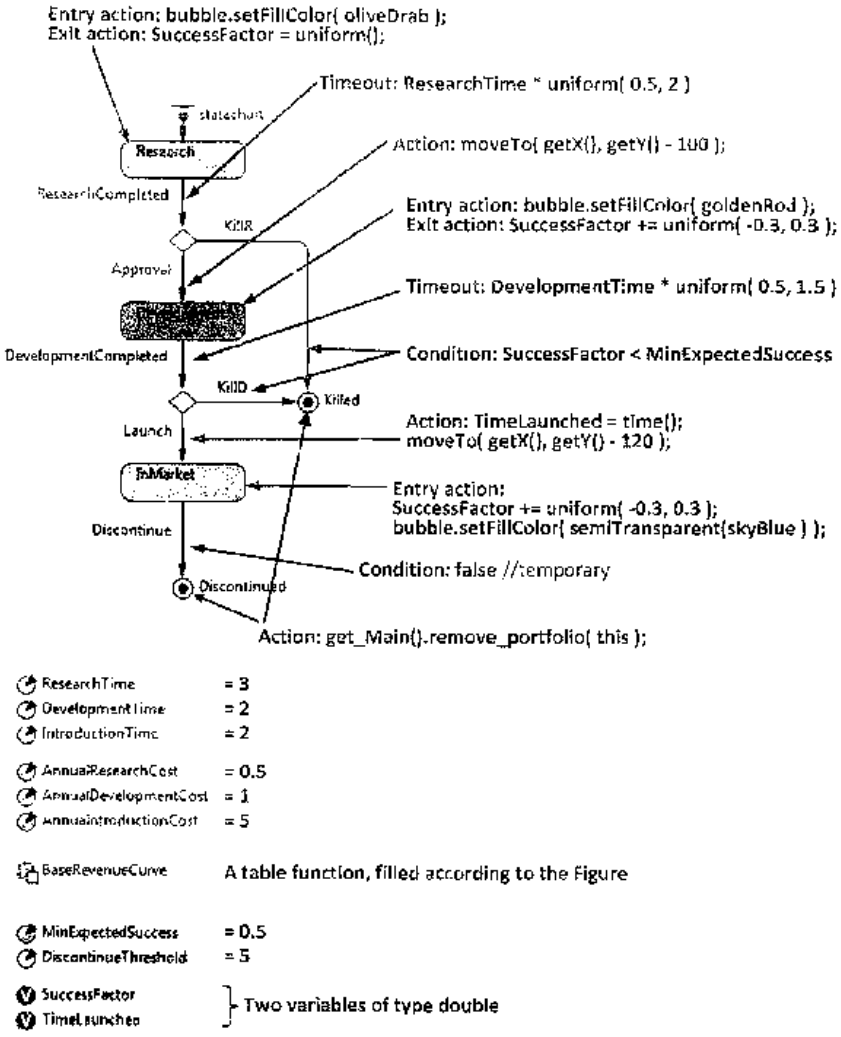


Figure 6.26 Statechart, variables, and parameters of the agent Product

In the beginning of the simulation, you can see 100 olive-colored bubbles scattered randomly. After a while, some bubbles turn brown and move up, and some disappear; these are the projects that were killed after the research phase. Then, more bubbles disappear, and the rest turn blue and move further up – these are the products that go to market. As the condition of the Discontinue transition is at the moment set to false, the products will remain in the market forever. We will now

add the cost and revenue calculation to the Product agent, fill in the missing condition, and enhance the animation.

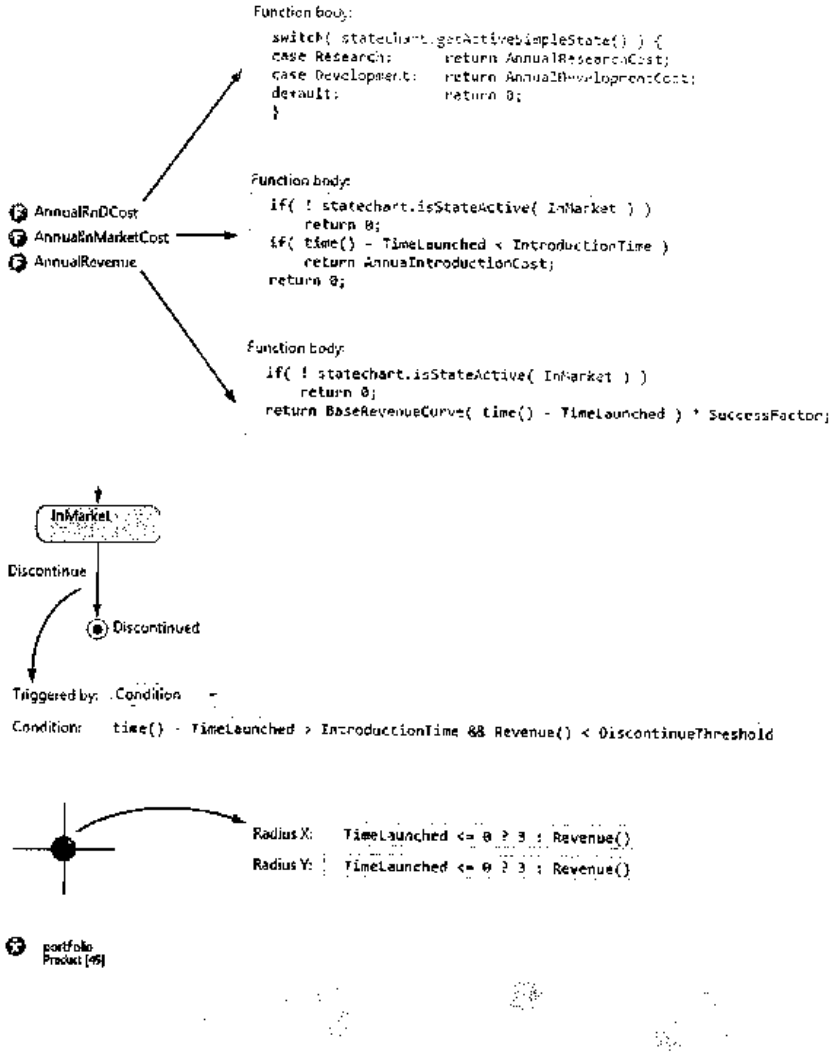


Figure 6.27 Functions calculating cost and revenue. Updated statechart and animation

▶ **Add the cost and revenue calculations, add the discontinue condition:**

8. In the editor of **Product**, add three functions with return type **double** as shown in Figure 6.27.

These functions calculate the cost and revenue based on the current state of the product. They use the statechart functions **isStateActive()** and **getActiveSimpleState()** to find out the state. However, while the product is in market, there is one special state that is not reflected in the statechart, namely this is the introduction phase that lasts for two years, according to our specification. To find out whether the product is in the introduction phase, we compare the time from the product launch (**time() – TimeLaunched**) and **IntroductionTime**. The time from the launch is also provided as an argument to the table function **BaseRevenueCurve()**.

9. Change the condition triggering the transition **Discontinue** as shown in Figure 6.27. The condition reflects the fact that the product can be discontinued only after the introduction phase.
10. Select the **bubble** (the animation of the product), and open the **Dynamic** page of its properties. Provide the expressions for the bubble radius as shown in Figure 6.27.
11. Run the model. Now you can see that bubbles of the products in market change their sizes dynamically as their revenue rises and then falls. Eventually, the bubbles seem to disappear.

The bubbles should disappear, because, according to the problem statement, the products have a limited market lifetime. However, you may notice that, although there are no bubbles at the end, the **portfolio** population still contains products. If you go inside any of the remaining products, you can see they are still in the **InMarket** state, despite their bringing in no revenue (and their bubbles have zero size!). The **Discontinue** transition does not seem to work: no products proceed to the **Discontinued** state, where it deletes itself from the model.

Why is this happening? To understand it, we need to recall how the trigger of condition type (see Section 7.3) works. The condition is tested once when the statechart enters the **InMarket** state (at this time, it obviously evaluates to false). Then *it will be re-evaluated if something happens within the agent*, for example, an event occurs, or somebody calls the agent's function **onChange()**. Also, *the condition is constantly monitored if the model contains dynamic variables.* Our model does not have an SD component yet, and nobody is telling the products to check if their revenue has fallen below the threshold.

12. To finish the current phase and to see the product deleting itself from the model, open the editor of **Main** and add any dynamic variable from the **System Dynamics** palette, say, **Stock**. This is, of course, a temporary solution.
13. Run the model again. Now you can see the **portfolio** is depopulated and then disappears.

The next step is to model the company investment policy. This will be done at the **Main** level. We will create statistics items in the **portfolio** agent population calculating the total revenue and costs, and use the statistics in the system dynamics model of investment. After that, we will model the start of new R&D projects, which depends on the money accumulated in one of the SD stocks.

▶ **Add statistics items to the product portfolio:**

14. Open the editor of **Main**, and select the **portfolio** population. Open the **Statistics** page of the population properties.
15. Create three statistics items (all three are of type **Sum** and have no condition):
    AnnualRevenue – the sum of **item.AnnualRevenue()**
    AnnualRnDCost – the sum of **item.AnnualRnDCost()**
    AnnualInMarketCost – the sum of **item.AnnualInMarketCost()**
16. Create the fourth statistics item that will count the number of products that are in R&D phase:
    NinRnD – the count of **!item.statechart.isStateActive(item.inMarket)**
    (it is easier to write that the product is not in market than to write that it is either in the research or in the development phase).
17. Delete the dynamic variable that you created in step 12 as a temporary solution. It is no longer needed.
18. Drag a regular dynamic variable from the **System Dynamics** palette, and call it **Revenue**. In the formula of the variable, type: **portfolio.AnnualRevenue()**.

You may remember that we did not recommend using the agent statistics functions directly in the formulas of dynamic variables, because the calculation may be time consuming. In this particular case, however, the number of products is not high, and we can afford it.

19. Run the model and watch how the value of **Revenue** changes over time, see Figure 6.28.

As one can expect, the total revenue of several products launched approximately at the same time is similar to the base revenue curve of a single product. The company is not investing in new product research and development, and in about 15 years it goes out of business.

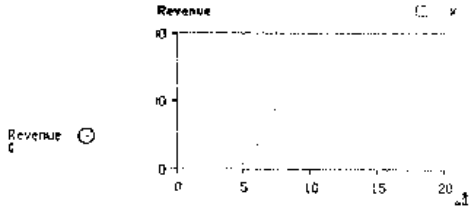


Figure 6.28 Total revenue of several products launched at approximately same time

▶ **Create the first draft of the stock and flow diagram:**

20. Starting with the **Revenue** variable created in the previous step, draw the stock and flow diagram as shown in Figure 6.29.
21. Run the model.

As you can see, shortly after the model starts, the **InvestmentCapital** stock falls down below zero, see Figure 6.29. It happens because at the beginning of the simulation the company starts too many (100) R&D projects simultaneously; they consume money, but bring in no revenue. As the products are launched in the market, the stock goes back up and then follows the S-shaped curve typical for systems with saturation. The **MainCapital** stock has a slight depression during the products' introduction period and then follows the same S-shaped curve.

In the next step, we will close the loop and implement the rule for starting new products, depending on the available investment money.

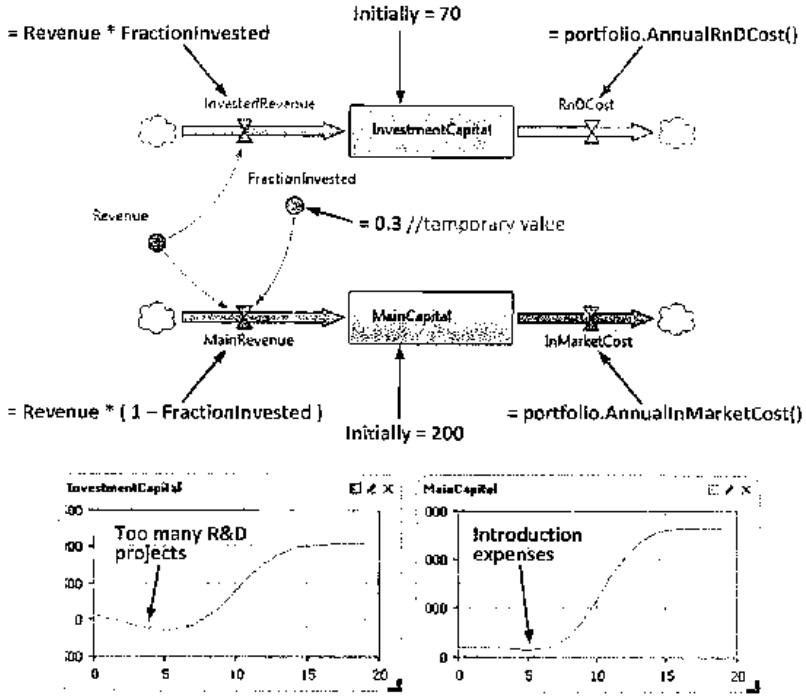


Figure 6.29 The first draft of the system dynamics model of investment policy

▶ **Create a condition event that will start new products:**

22. In the editor of **Main**, create three parameters and a condition-triggered event **StartNewProject** as shown in Figure 6.30.
23. Select the **portfolio** population, and set its **Initial number of objects** to 0.
24. Run the model.

The parameter **AverageProjectCost** is an estimation of how much money will be required (per project) to finish all the ongoing projects plus a new one. The event **StartNewProject** is constantly monitoring the **InvestmentCapital** stock and, when it detects room for one more project, starts it. The one-time project setup cost is immediately subtracted from the stock, and a new **Product** agent is created in the **portfolio** population. The last statement in the event action (the call of **restart()** function) tells the event to resume monitoring the condition after each occurrence. Because new products are now created automatically, we can set the initial number of products to 0.

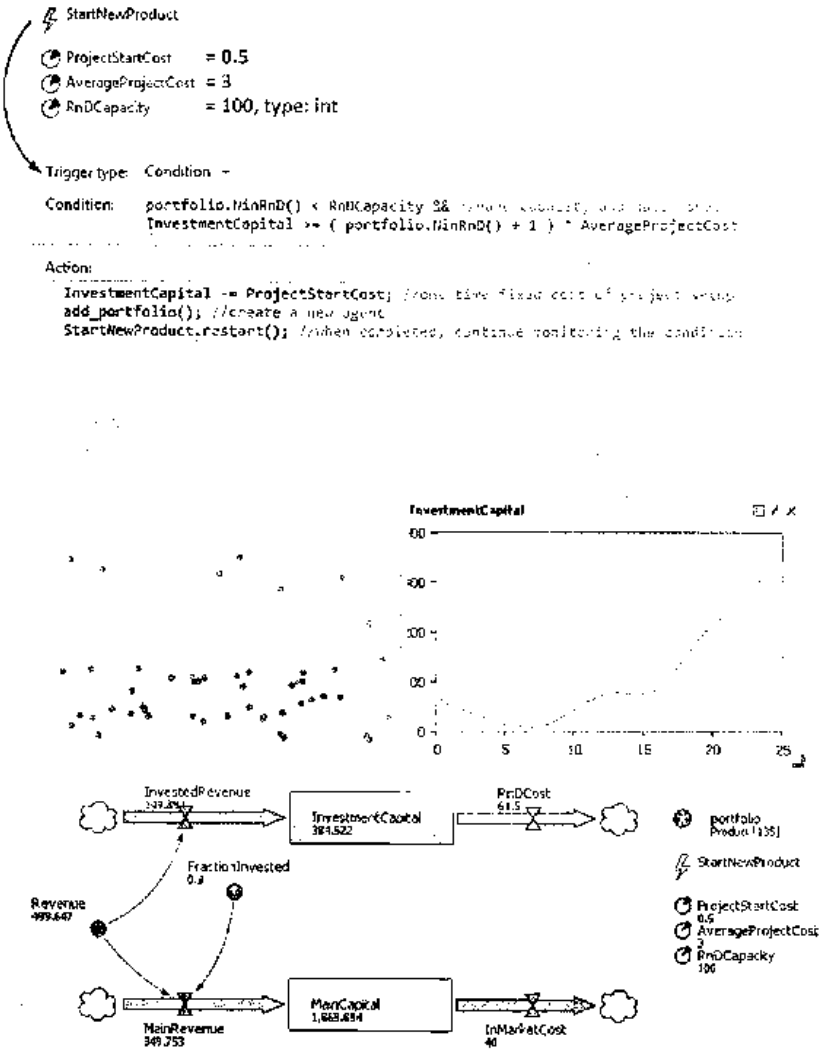


Figure 6.30 Condition event that creates new products. The new dynamics of the model

Now, company acts safely and is profitable. The continuous R&D process ensures that the company always has new products to offer. The revenue stream grows during the first three decades up to approximately $800M/year and then starts oscillating irregularly around that value. Further revenue growth is limited by the R&D capacity of the company.

Under the current model setup, 30% of the company gross revenue always goes into the investment capital stock, which continues to build up and remains largely unused. In the last step, we will implement the remaining part of the investment policy, namely, we will make the invested fraction of the revenue a variable that depends on the accumulated resources. This is done purely in the system dynamics part of the model.

▶ **Modify the formula for the invested fraction of the revenue:**

25. Add one more parameter: **MaxFraction** = 0.2.
26. Change the formula of **FractionInvested** to
    investmentCapital > RnDCapacity * AverageProjectCost ? 0 : MaxFraction
27. Run the model.

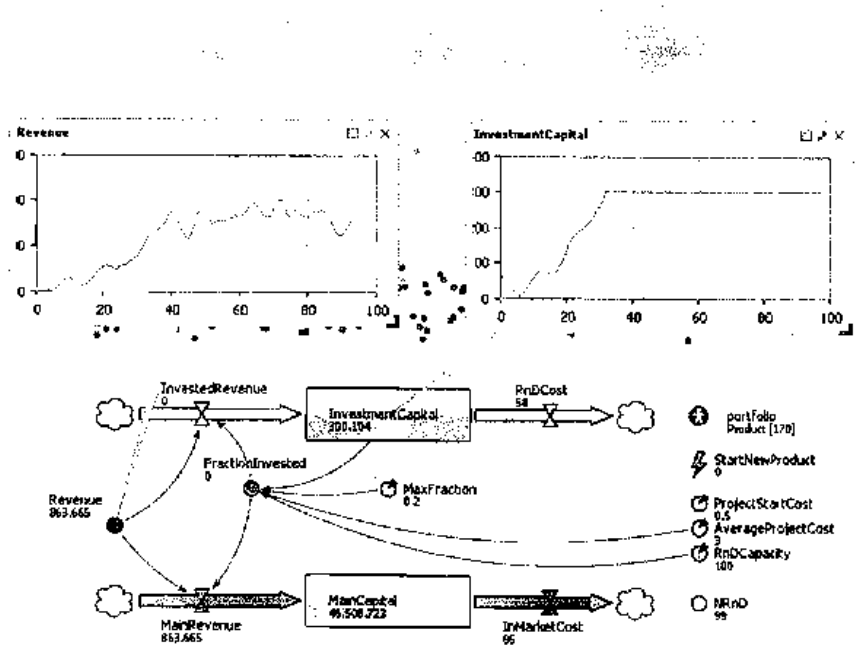


Figure 6.31 Total company dynamics under the fully implemented investment policy

The picture is different. The **investmentCapital** stock reaches the value of $300M and stops growing. The revenue oscillates around $800M/year, which, as we know already, is the upper limit with the given R&D capacity.

We can use this model to optimize the investment policy. For example, we can investigate how sensitive the company dynamics are to the parameters of the

investment policy, say, to **MaxFraction**. We will compare the revenue over time curves obtained in different simulation runs.

▶ Create a sensitivity analysis experiment:

28. In the editor of **Main**, right-click the **Revenue** variable and choose **Create data set** from the context menu. A dataset **RevenueDS** is created. In the dataset properties, select **Update data automatically**.
29. Right-click the model (topmost) item in the **Projects** tree, and choose **New | Experiment** from the context menu. The experiment wizard opens.
30. In the first page of the wizard, select **Sensitivity analysis**.
31. In the **Parameters** page of the wizard, select **MaxFraction** as the varied parameter and specify **Min:** 0.01, **Max:** 0.4, **Step:** 0.01.
32. In the **Charts** page of the wizard, fill in one row of the table: **Title:** Revenue over time, **Type:** dataset, **Expression:** root.RevenueDS. Click **Finish**.
33. A sensitivity analysis experiment is created. Right-click it in the **Projects** tree, and choose **Run**.

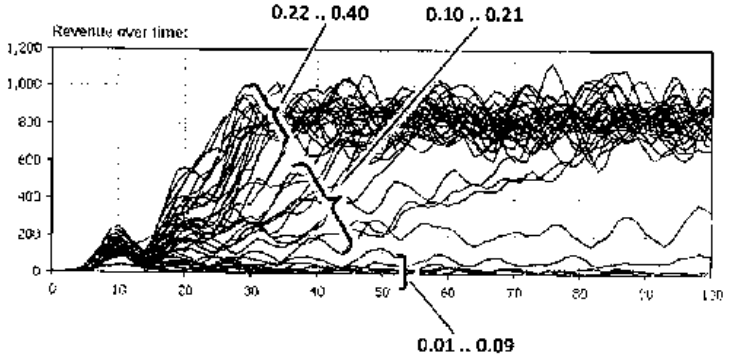


Figure 6.32 Sensitivity analysis: revenue dynamics under different invested fraction values

The results are interesting. The revenue over time curves cluster into three groups, see Figure 6.32. In the first one the company goes out business; this corresponds to the values of **MaxFraction** from 0.01 to 0.09. When the parameter is in the range 0.10..0.21, the revenue climbs up – the higher the **MaxFraction**, the faster it reaches the maximum value of $800M. Further increase of the invested revenue fraction does not affect the growth.

6.4. Discussion

When developing a discrete event model of a supply chain, IT infrastructure, or a contact center, the modeler would typically ask the client to provide the arrival

rates of the orders, transactions, or phone calls. He would then be happy to get some constant values, periodical patterns, or trends, and treat arrival rates as variables exogenous to the model. In reality, however, those variables are outputs of another dynamic system, such as a market, a user base. Moreover, that other system can, in turn, be affected by the system being modeled. For example, the supply chain cycle time, which depends on the order rate, can affect the satisfaction level of the clients, which impacts repeated orders and, through the word of mouth, new orders from other customers. The *choice of the model boundary* therefore is very important.

The only methodology that explicitly talks about the problem of model boundary is system dynamics. However, the system dynamics modeling language is limited by its high level of abstraction, and many problems cannot be modeled with the necessary accuracy. With multi-method modeling you can choose the best-fitting method and language for each component of your model and combine those components while staying on one platform.

? A telecom company is about to introduce a new type of service, say, HDTV or high-speed Internet access and is planning the additional network infrastructure, the tariff policy, and the marketing campaign. Model adoption of the new technology by the users in the loop with the network infrastructure performance. Consider potential dissatisfactions effects, incremental growth of the infrastructure capacity, and ROI.

Chapter 7. Designing state-based behavior: statecharts

7.1. What is a statechart?

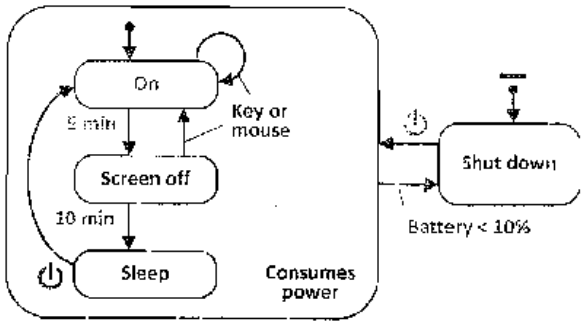
A *statechart* (an extended version of *state diagram*) is a visual construct that enables you to define event- and time-driven behavior of various objects. Statecharts are very helpful in simulation modeling. They are used a lot in agent based models, and also work well with process and system dynamics models.

Statecharts consist of *states* and *transitions*. A state can be considered as a "concentrated history" of the object and also as a set of reactions to external events that determine the object's future. The reactions in a particular state are defined by transitions exiting that state. Each transition has a *trigger*, such as a message arrival, a condition, or a timeout. When a transition is taken ("fired") the state may change and a new set of reactions may become active. State transition is atomic and instantaneous. Arbitrary actions can be associated with transitions and with entering and exiting states.

AnyLogic supports a version of UML statecharts, ("UML state machine", n.d.)which, in turn, is the adapted version of David Harel statecharts. AnyLogic statecharts have composite states (states that contain other states), history states, transition branching, and internal transitions. Orthogonal states are not supported, but you can define multiple statecharts for an object that will work in parallel.

Example 7.1: A laptop running on a battery

Let us consider an example: a laptop running on a battery. When the laptop is on and the user is working, i.e. is pressing the keyboard keys and moving the mouse, the laptop stays in the **On** state, see Figure 7.1. However, after 5 minutes of user inactivity, the laptop turns off the screen to save power. This fragment of the laptop behavior is modeled by two transitions exiting the **On** state: a timeout transition **5 min** and a loop transition triggered by **Key or mouse** (although the loop transition does not change the statechart state, it resets the timeout: 5 minutes are calculated since the last entry to the **On** state, i.e. last user action). If the user touches the keyboard or mouse while the screen is off, the screen turns back on, hence the **Key or mouse** transition from **Screen off** back to **On**. If, however, the user remains inactive for 10 minutes more, the laptop switches to the **Sleep** mode to further minimize the battery usage. This is modeled by the transition **10 min**. In the **Sleep** state, the laptop does not react to the mouse and the keyboard; therefore, there are no corresponding transitions from that state. To wake the laptop up again, you need to press the power button ⏻, which triggers the transition from **Sleep** to **On**.



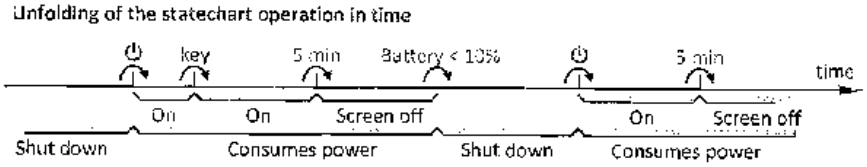


Figure 7.1 Statechart of laptop running on battery

In any of the three states (**On**, **Screen off**, or **Sleep**) the laptop consumes the battery power. When the battery level falls to 10% the laptop is forced to shut down, regardless of the state. How do we model this? We, of course, can draw three transitions (one from each state) triggered by a condition **Battery < 10%**, and this is what we would do if we were using a classical "flat" state machine. However, statecharts offer a much better way to define such "interrupt-like" reactions common to a group of states: you can draw a *composite state* around the group and define reaction at the composite state level. The transition **Battery < 10%** exiting the composite state **Consumes power** will bring the statechart to the **Shut down** state no matter in which of the three inner *simple states* it occurs.

In the **Shut down** state the laptop will only react to the power button, so there is only one transition from there to **Consumes power**. As the latter is a composite state, we need to specify the initial simple inner state where the statechart gets after entering the composite state. In our case this is **On**, so **On** is marked with the *initial state pointer*. Similarly, we need to mark the initial state at the topmost level of the statechart, either **Consumes power** or **Shut down**. You can see the *statechart entry point* that points to the **Shut down** state.

How do statecharts differ from action charts and flowcharts?

In simulation modeling in general, and in AnyLogic in particular, we use other diagrams that look similar to statecharts (have boxes connected with arrows), but

have different meaning, or semantics. For instance: action charts, process flowcharts, and stock and flow diagrams (see Section 5.1). It is worth considering the difference between these diagrams:

Statecharts define internal states, reactions to external events, and the corresponding state transitions of a *particular object*: a person, a physical device, an organization, a project, etc. The (simple) states of the statechart are *alternative*: at any given moment in time the statechart is in *exactly one simple state*.

Action charts are a graphical way to define algorithms or calculations. The arrows define the control flow for a calculation. The action chart does not persist in time: it executes logically instantaneously from start to end, and it does not have a notion of state. Action chart can be considered as a graphical form of a Java function.

Process flowcharts (composed of the Enterprise Library objects) define sequences of operations performed over entities. During the simulation there may be multiple entities in different places of a flowchart, so the "state" of the flowchart is actually spread across the whole diagram.

Stock and flow diagrams are a fundamental part of system dynamics (see Chapter 5) models. Stocks are accumulations and the state of the system is defined by the values of all stocks. Compared to statecharts, all stocks and all flows are simultaneously active.

7.2. Drawing statecharts

The statechart building elements are located in the **Statechart** palette, see Figure 7.2. You can drag them on the canvas. States and transitions also support drawing mode (explained below). While you draw a statechart, its structure is permanently validated and inconsistencies found are displayed in the **Problems** view.

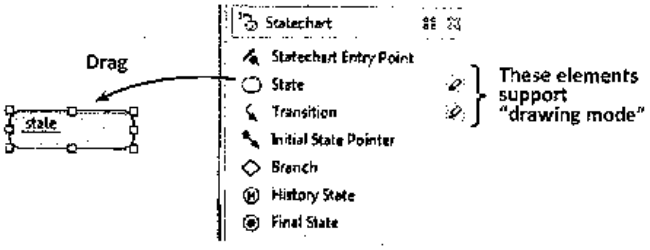


Figure 7.2 The Statechart palette

Simple states

Drawing a statechart typically starts with drawing states, giving them names, and adjusting their sizes and colors.

▶ **To create a state in the drawing mode:**

1. Double click the **State** icon in the **Statechart** palette, or double click the pencil icon on the right hand side of the **State** icon.
2. Click and hold on the canvas where the upper left point of the state should be placed.
3. Drag the cursor to the bottom right point of the state.
4. Release the mouse button.

The in-place editor of the state name opens automatically just after you create a state. And, of course, you can change the state name any time later.

▶ **To change the state name:**

1. Double-click the state name in the graphical editor.
2. Type the new name in the in-place text editor.

Alternatively, you can change names of statechart elements in the properties window. The position of the state name relative to the state can be adjusted.

▶ **To adjust the position of the state name:**

1. Select the state by clicking it.
2. Drag the state name.

The statechart can be made more informative if you paint its states with different colors.

▶ **To change the color of the state:**

1. Select the state.
2. Choose a new color in the **Fill color** field of the state properties.

Transitions

You can drag the **Transition** object on the canvas, connect it to states, and edit its shape, or you can draw a transition point by point in the "drawing mode".

▶ **To draw a transition point by point (in the drawing mode):**

1. Double click the **Transition** icon in the **Statechart** palette.
2. Click in the canvas at the position of the first point of the transition.
3. Continue clicking at all but the last points.
4. Double click at the last point position.

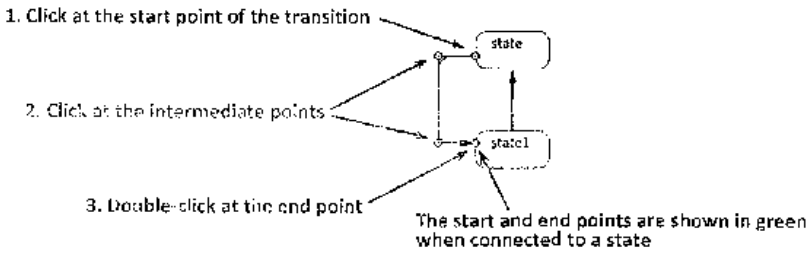

1. Click at the start point of the transition
2. Click at the intermediate points
3. Double-click at the end point
state
state1
The start and end points are shown in green when connected to a state

Figure 7.3 Drawing a transition point by point

After a transition has been created, you can edit its shape by moving, adding, or removing points.

▶ **To edit the transition shape:**

1. Select the transition.
2. To add a new point double click the edge where the point has to be added.
3. To delete a point double click it.
4. To move a point, drag it to the new position.

A transition gets automatically connected to a state if its start or end points lie on the state boundary. You can check if the transition is connected by selecting the transition and looking at the start and end points: when connected, the point is highlighted with green. If a point is not connected to a state, a "Hanging transition" or "Element is not reachable" error is displayed in the **Problems** view. You can click on that error message and the transition will be displayed and selected. To fix the error drag the point to the state boundary.

By default, the transition name is not displayed in the graphical editor and at runtime. You can change this.

▶ **To display the transition name:**

1. Select the transition.
2. Check the checkbox **Show name** in the transition properties.

In most cases you would need to adjust the position of the transition name. To do this, drag the name while the transition is selected.

Statechart entry point

Each statechart *must have exactly one* statechart entry point – an arrow that points to the initial top level state (a top level state is a state that is not contained in any composite state). You may have noticed that until you add the entry point, the error "Element is not reachable" is displayed for all statechart elements. The errors will disappear once you create an entry point.

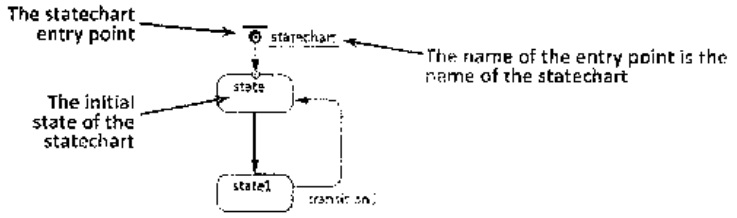

The statechart entry point
statechart
The name of the entry point is the name of the statechart
state
The initial state of the statechart
state1

Figure 7.4 The statechart entry point

▶ **To create a statechart entry point:**

1. Drag the **Statechart entry point** from the **Statechart** palette to the canvas.
2. Drag the end point of the arrow to the boundary of the initial top-level state. The end point will show in green when connected.

The name of the statechart entry point is the name of the statechart.

Do not confuse the statechart entry point with the initial state pointer. The latter is used to mark the initial state within a composite state.

Composite states

A composite state is created in the same way as a simple state. You can drag the **State** on the canvas and then resize it so that it contains other states, or you can draw it around other states in the drawing mode. The graphical editor prevents you from drawing intersecting states, so the state structure is always strictly hierarchical.

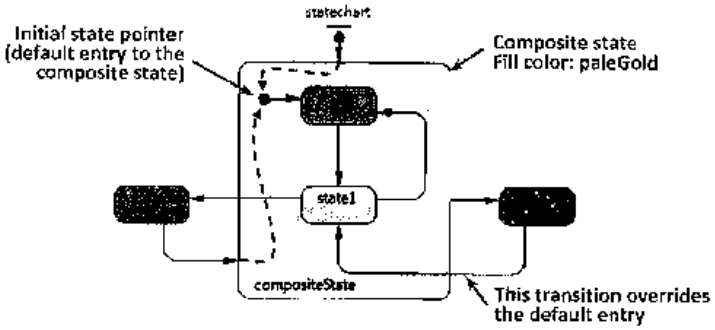


Figure 7.5 Composite state

When a state becomes composite, its fill color is set to transparent. You can change it by setting the **Fill color** field in the state properties. In most cases (namely, each time there is a transition or a pointer pointing to the composite state), you need to identify the default initial state inside the composite state, i.e. one level down in the

state hierarchy. This is done by the initial state pointer that can be found in the same **Statechart** palette.

Transitions may freely cross the composite state boundary both inbound and outbound. In particular, an inbound transition can lead to an internal state other than the default initial state (see Figure 7.5).

History state

History state is a pseudo-state – it is a reference to the *last visited state inside a composite state*. History states are used to model a return to the previously interrupted activity.

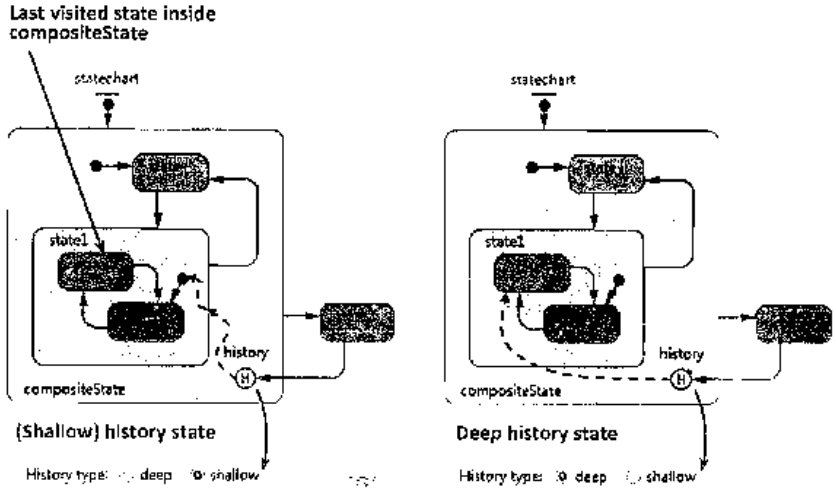


Figure 7.6 History state

History state can be located only inside a composite state. There are two types of history states: *shallow* and *deep*. Shallow history state remembers the last visited state on the same level of hierarchy, and deep history state remembers the last visited simple state, which may be at a deeper level. You can switch between shallow and deep in the history state properties.

Consider the statechart in Figure 7.6. Assume the statechart was in **state4** when the transition took it to **state2**. The transition from **state2** points to the history state; therefore, when entering the **compositeState** that way the statechart will return to the last visited state within the **compositeState**. If the history is shallow (on the left) it refers to **state1**, which is on the same level as the history state. Therefore, the statechart will enter **state1** and then go to the default state inside **state1** (**state3**). If the

history is deep, the statechart will fully restore its state on the moment of leaving the compositeState, (returning to state4).

Final state

Final state is a state that terminates the statechart activity. Final state may be located anywhere – inside composite states or at the top level. There may be any number of final states in a statechart. A final state may not have any outgoing transitions.

Consider the statechart in Figure 7.7. The final state is inside state1, and upon entering it, the statechart deactivates all transitions that were active.

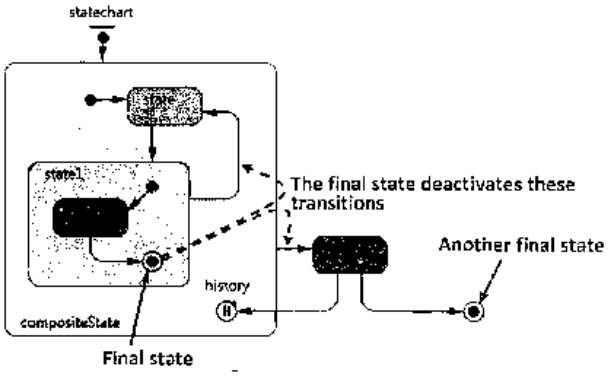
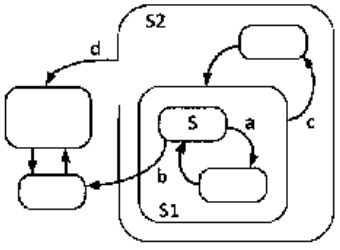


Figure 7.7 Final state

7.3. State transitions: triggers, guards, and actions

Which transitions are active?

As you know, the transitions from the current state of the statechart define how the statechart will react to the external events and conditions. Consider Figure 7.8.



S is the current active simple state

S1 and S2 are active composite states

All transitions from S, S1, and S2 are active: a, b, c, and d

Figure 7.8 Active states and active transitions

At any moment during the statechart lifetime there is exactly one current (or *active*) simple state. In our case, this is **S**. **S** is located inside the composite state **S1**, which, in turn, is inside **S2**. Both **S1** and **S2** are also active states.

You can test whether a state is active by calling the statechart method **isStateActive(<state name>)**. You can find out the current simple state by calling **getActiveSimpleState()**.

All transitions exiting any active state are active (they may be triggered). In the case of the statechart in Figure 7.6, these are transitions **a**, **b**, **c**, and **d**.

Trigger types

In the Table below we explain all possible ways to trigger a statechart transition, what different trigger types are used for, and how they work.

Trigger type	Primary use	Transition fires
Timeout	**Timeout**: change state if other awaited events do not occur within the specified time interval. **Delay**: stay in a state for a given time, then leave.	After a specified time interval counted from the moment the statechart enters the direct "source" state of the transition (i.e. the state on whose boundary the transition start point is located). The timeout expression is calculated one time when the statechart enters that state. The expression can be stochastic as well as deterministic.
Rate	Sporadic state change with a known mean time. In agent based models used to represent sporadic decisions made by an agent under a certain, possibly variable, influence (purchase decisions, adoption of ideas, etc.).	Same as timeout, but the time interval is drawn from an exponential distribution parameterized with the given rate. For example, if the rate is 0.2 the timeouts will have mean values of 1/0.2 = 5 time units. If a change occurs in the active object while the rate transition is active (namely, if **onChange()** method is called), the rate expression is re-evaluated and, if it gives a new result, the transition is rescheduled using a new exponential distribution.

Condition	Monitor a condition and react when it becomes true. For example: buy if the stock price falls below a certain threshold; launch a missile if the aircraft is closer than 5 miles, etc.	Once a given condition becomes true. The condition is an arbitrary boolean expression and may depend on the states of any objects in the whole model with continuous as well as discrete dynamics. In most cases you can assume the condition is constantly monitored while the transition is active. For more information see Section 8.2.
Message	React to messages received by the statechart or by the active object from outside. The messages can model communication between people or organizations, commands given to a machine, physical products, electronic messages, etc.	Upon reception of a message that matches the template specified in the transition properties. The ways of sending a message to a statechart as well as the rules of message processing are described below.
Arrival	React to arrival. Can be used in agents moving in 2D or 3D continuous space.	When the agent arrives at the destination point (assuming its movement was initiated by calling **moveTo()** method). This is implemented as a message of a special type sent to the statechart by calling the method **fireEvent()**

Timeout expressions

The expression in the **Timeout** field of the transitions triggered by timeouts is interpreted as time interval in model time units. If you write **10** there, it may mean 10 seconds, 10 days, 10 weeks, etc, subject to the time unit specified in the experiment properties. To make the timeout expression independent of the model time unit, you should use the functions like **millisecond()**, **second()**, **minute()**, ..., **day()**, **week()**. For example, the expression **10*day()** always evaluates to 10 days, regardless of the experiment settings. More information about the model time, time units, and usage of calendar can be found in Chapter 16, "Model time, date and calendar. Virtual and real time".

For timeouts measured in time units larger than a week, and also for correct handling of daylight saving time (see Section 16.2), you should use the function:

```
double toTimeout( int timeUnit, int amount )
```

where timeUnit can be YEAR and MONTH as well as WEEK, DAY, etc. That function returns the time interval between the current time and the time in the future, which is the amount of given time units later. *If you wish a transition to fire at the same time and date but 15 years later than the statechart comes to a state*, the transition trigger expression should be:

```
toTimeout( YEAR, 15 )
```

Sometimes you want the transition to fire *15 years later than a given date* (typically in the past) that you stored in the lastDate variable of type Date. In this case, the timeout expression should be:

```
dateToTime( lastDate ) + toTimeout( YEAR, 15 ) – time()
```

Here we first convert the lastDate into model time, then add 15 years timeout and subtract the current time (remember that the expression is calculated when the statechart enters the transition's source state). If you want the timeout transition to fire *exactly at 9AM on the nearest Monday*, you can create and use the function described in Section 16.2:

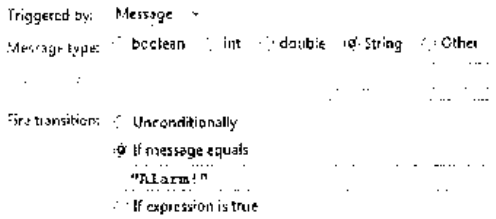
```
timeOnNearestDayOfWeek( MONDAY, 9, 0, 0 ) - time()
```

The same applies to timeout events (see Section 8.2).

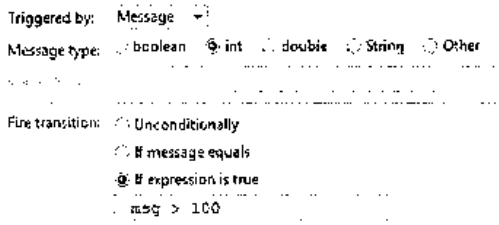
Transitions triggered by messages

You can send messages to a statechart, and the statechart can react on message arrival. A message can be an arbitrary object: a string, a number, an ActiveObject, etc. You can set up the transition to check the message type and content. Several examples of transition triggered by a message are given in Figure 7.9.

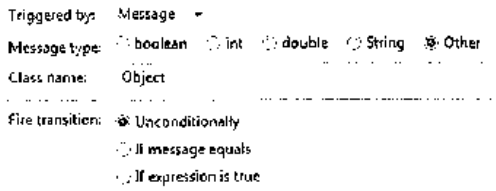
Transition triggered by the string "Alarm!"



Transition triggered by an integer that is greater than 100



Transition triggered by any message at all



Transition triggered by an active object whose name starts with "Company"

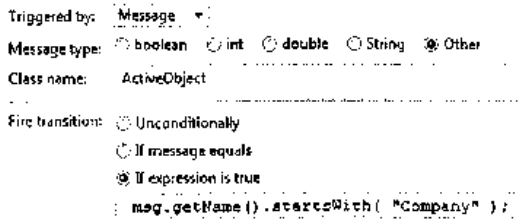


Figure 7.9 Examples of transitions triggered by messages

The message type is checked first. The properties of the transition allow specifying of several standard types (numeric – **int** or **double**, **String**, **boolean**) or a custom type in the field **Class name**. **Object** is the most general type; if **Object** is specified, any message

will pass the type filter. After the type check, the transition is triggered by the message either unconditionally, or if it equals the given object, or if a given expression evaluates to true. In the expression, you can refer to the message as `msg` (see the last example in Figure 7.9). Some recommendations on using the message types are given in the Table below.

Message type	Primary use
`String`	There is a finite number of notifications you wish to send to a statechart and the messages do not carry any additional data. For example, a statechart that models a coffee machine may react to the following message set: `"Espresso"`, `"Cappuccino"`, `"Latte"`.
Numeric (`int` or `double`)	You need to notify a statechart about a certain quantity and the statechart always knows what this quantity means. For example, a statechart that models a behavior of a broker who watches the price of a particular stock can receive price updates in the form of double type messages
Object of custom type	Messages contain several pieces of information for example: the sender's address, the destination address, the command id, the parameters, etc. For such messages, you would typically define a new Java class (see Section 10.11) with several fields.

Sending messages to a statechart

A message can be sent to a statechart by calling either the method `fireEvent()` or the method `receiveMessage()`; the ways the messages are then processed by the statechart are different.

If `receiveMessage( <message> )` is called, the statechart looks through its currently active transitions and, if the message matches a transition trigger, the transition is scheduled for execution. If the message matches more than one transition, all matched transitions will be scheduled, but later on firing of one transition may cancel the others. If the message does not match any of the triggers, it is discarded.

If the message was sent to a statechart by `fireEvent( <message> )`, it is added to the *message queue* and, at the same time, the statechart tries to match each active transitions with *each* message in the queue. If a match is found, the message is consumed by the transition and all messages before it are discarded. The queue does not persist in time: *if the model time is advanced, all unconsumed messages are lost*. However, the statechart may make several steps taking zero time reacting to several messages. Figure 7.10 shows how the message queue is handled. Initially, all messages came to the statechart at the same model time `t`. The messages `x` and `y`

were discarded because they did not match any transition. The message c was discarded because non-zero time elapsed since it has been added to the queue.

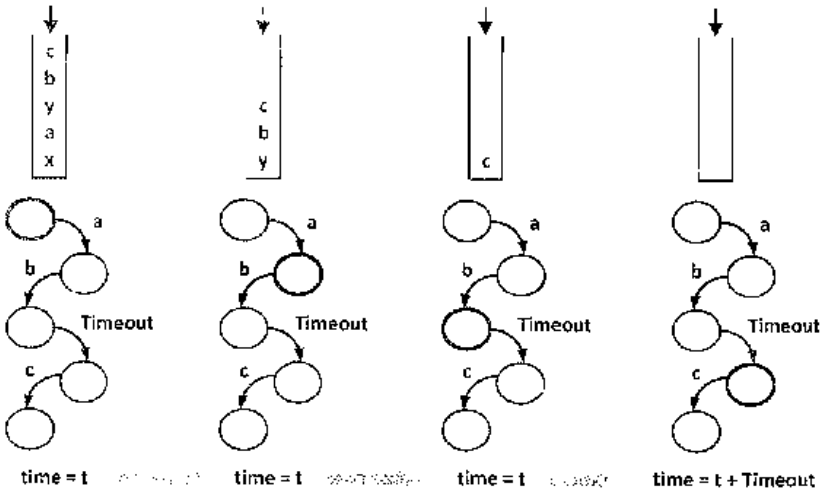


Figure 7.10 Statechart message queue

If the statechart is located inside an agent, the messages received by the agent (see Section 3.8) can be forwarded to the statechart. When forwarding, the agent uses the **fireEvent()** method.

▶ **To forward the agent's incoming messages to statechart(s):**

1. Open the **Agent** page of the **Properties** of the active object (agent).
2. In the **Forward message to** section, select the statecharts where you wish to forward the messages.

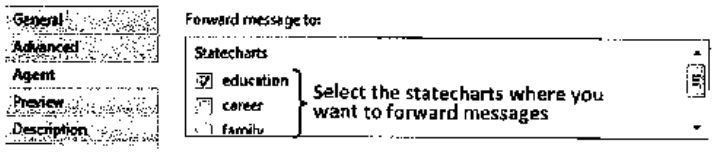


Figure 7.11 Forwarding the agent's incoming messages to statecharts

Guards of transitions

When the transition is ready to fire, it performs one final check – it checks its *guard* and, if the guard evaluates to false, the transition is *not taken*. Thus, guards can impose additional conditions on state transitions. Guard is an arbitrary boolean expression and can refer to any object in the model.

Consider an example: a car engine that would not start if the gearbox is not in the Parked position. The model of the car can have two statecharts (see Figure 7.12): one for the engine and another for the gearbox. The transition start of the engine triggered by the "Start/Stop" button will only be taken if the gearbox statechart is in the Parked state.

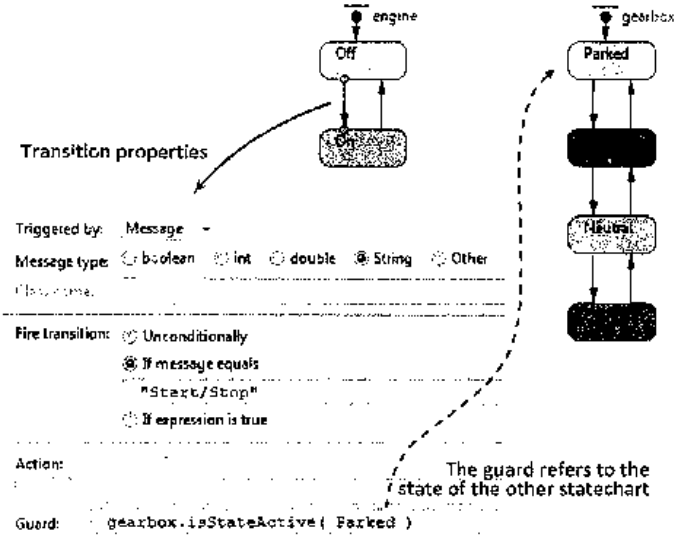


Figure 7.12 Using guards: car engine state transition depends on the gearbox state

If the transition was ready to fire but did not fire because the guard evaluated to false, then:

- **Timeout** transitions get deactivated
- **Rate** transitions get deactivated until the rate is changed
- **Condition** transitions continue monitoring the condition
- Transitions triggered by a **message** continue waiting for another message
- Transitions triggered by **arrival** continue waiting for the next arrival

Do not confuse guards and conditions. The condition that triggers the transition is constantly monitored while the transition is active. The guard is only checked when the transition is ready to fire, i.e. when its trigger is available.

Transitions with branches

A transition may have *branches*, and can bring the statechart to different states depending on conditions. Those conditions are evaluated and the target state is

chosen immediately after the transition has been triggered and its guard evaluated to true, so firing of a transition with branches can be considered as instant and atomic.

The number and configuration of decision points and branches can be arbitrary. Each decision point is created using the **Branch** object from the **Statechart** palette. The branches after the decision points are created and edited in the same way as transitions, but their properties are different: they have **Condition** and **Action**. The conditions of branches exiting a decision point should be *complete and unambiguous*. If during the transition firing two or more conditions evaluate to true, one of the corresponding branches is chosen nondeterministically. If none of the conditions evaluate to true, a runtime error is signaled.

The statechart in Figure 7.13 is set up to test a number that it receives as a message. The transition from **WaitNumber** state is triggered by a message of type int. The transition branches into three: one for negative numbers, one for zero, and the third one for everything else (default). Obviously, a positive number will cause the statechart to follow the third branch, which, in turn, has two exits: one for even numbers and another (default) – for odd. The expression **X % 2** returns a remainder of **X** divided by **2** (see Section 10.5, "Expressions").

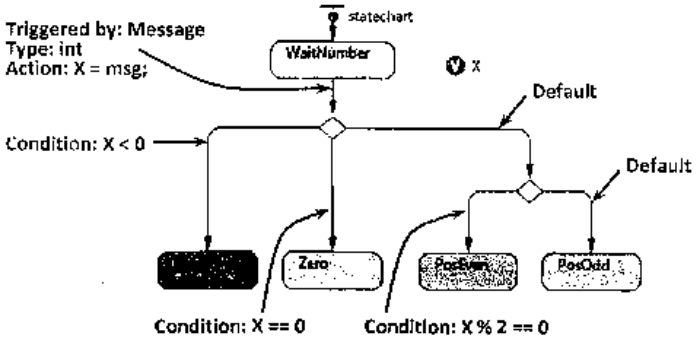


Figure 7.13 A transition with two decision points and five branches

Frequently, when a transition triggered by a message has branches, the conditions of the branches depend on the message content. As the message received is not available outside the first section of the transition where the trigger is defined, you may need to save the message in a variable, as in the example above.

Internal transitions

With the help of *internal transitions* you can define some activity the statechart performs while being in a state without exiting that state. Consider a mobile phone

user who is a frequent traveler. While abroad, he makes phone calls using roaming, and the frequency of those calls is less than the frequency of calls at the home location. We can model this behavior as a statechart with two states: AtHome and Abroad. Let the person spend from 2 to 5 weeks at home and from 3 to 7 days abroad. This is modeled by the timeout transitions flyOut and flyBack that limit the time the user stays in each of the two states.

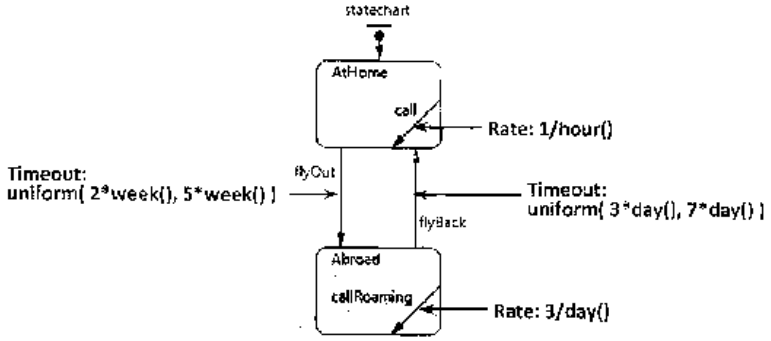


Figure 7.14 A statechart of a mobile phone user with internal transitions

At home, the person makes on average of one call per hour. We can model this by creating an internal transition call, that fires at the rate 1/hour(). Graphically, the transition call is *entirely inside* the state AtHome, which tells AnyLogic that the transition should be treated as internal and its firing does not cause exit and re-enter of the state AtHome. Therefore, the 2-5 week timeout is *not reset* by the phone calls. Similarly, the callRoaming transition does not affect the duration of the trip abroad defined by the transition flyBack.

The transition is counted as internal if it exits and enters the same state and all its intermediate points are inside that state.

Firing of an internal transition interferes neither with other internal transitions nor with other state transitions that may be defined inside the same state.

Order of action execution

All statechart elements (states, transitions, decision points, transition branches, etc) may have actions associated with them. The actions are executed as control is passed from one state to another along a transition. To understand the *order of action execution*, consider the statechart in Figure 7.15.

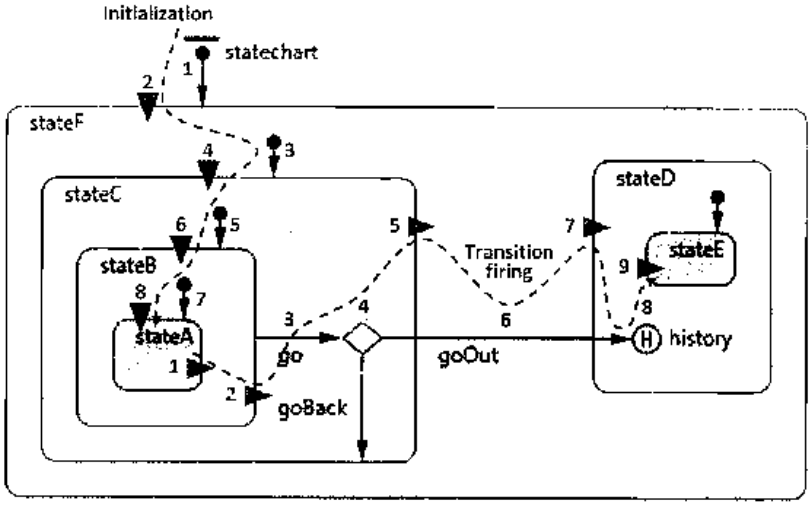


Figure 7.15 The order of action execution in a statechart

During the initialization, the statechart executes first the action of its entry point and then entry actions of the states and actions of initial state pointers down the state hierarchy all the way to the initial simple state (see the initialization path and action numbering in Figure 7.15).

During the transition firing, the statechart first exits the current simple state (in our case stateA), then all states up the state hierarchy to the transition source state (stateB). After that, the action of the transition is executed, and then the statechart enters the transition target state (in our case this is a history state, which points to stateE). If the target state is composite, the statechart follows the initial state pointers down to the next simple state. If the transition has decision points and branches, their actions are executed in the natural graphical order (i.e., they can be mixed with exit and entry actions of states (see the action numbering in Figure 7.15). If the transition with its source and target states is fully contained in a composite state (like stateF), that state and all states above it are not considered as exited or entered.

Synchronous vs. asynchronous transitions

Subject to the input data available, sometimes it makes sense to use *synchronous state transitions* (i.e., the transitions where the decision to change state is made in several attempts at regular time intervals - on "ticks"). Consider a model of alcohol use by an individual. Under a certain age, the person does not use alcohol at all, then

starts drinking. The input data may be in the form of a table where initiation probability is provided for each age, see the Table.

Age	Probability of alcohol initiation	Age	Probability of alcohol initiation
11	0.00	18	0.23
12	0.01	19	0.17
13	0.05	20	0.11
14	0.05	21	0.05
15	0.12	22	0.03
16	0.14	23	0.01
17	0.18	24	0.00

Suppose in the model there is a statechart with two states: **NeverUser** and **User**. How do we design a transition from one to another? Obviously, this can be a timeout transition with time drawn from a certain distribution. We do not have that distribution explicitly and could construct it from the data we have (which might be an interesting analytical exercise). However, instead we can create a synchronous transition and use the input data directly.

In Figure 7.16 the transition exiting the state **NeverUser** is taken every year (assuming the time unit (see Section 16.1) is year). Then the decision is made: go to the state **User** with the probability taken from the table function according to the current age, otherwise return to **NeverUser**.

The age in this model equals the model time as the statechart is created at time 0 and the time unit is year, hence time() is provided as an argument when accessing the table function. In general, this may be not true (for example, if the statechart belongs to an agent that may be created in the middle of the model runtime). In that case, you should remember the year the agent was born in the agent's local variable and calculate the age as time() - <year born>.

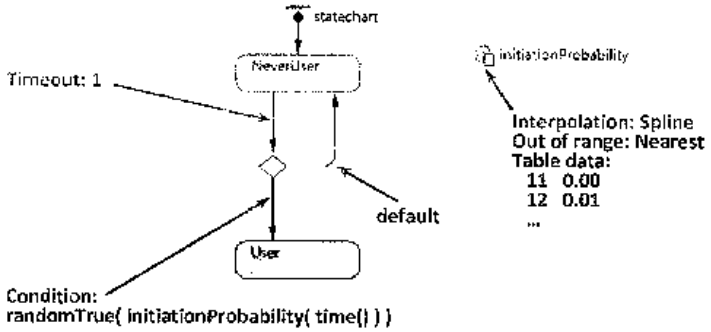


Figure 7.16 Alcohol initiation modeled by a synchronous state transition

The table function is set up in such a way that it returns the nearest value when the argument is out of range. Therefore, for ages under 11 and ages over 24, the initiation probability is zero .

7.4. Statechart-related API

The API related to statecharts is contained in the class **Statechart**, and also in the classes **ActiveObject** and **Transition**. These two methods send messages to the statechart. They are discussed in detail in Section 7.3.

- **boolean receiveMessage(Object msg)** – posts a message to the statechart without putting it to the queue. Returns **true** if the message matched the trigger of at least one transition, otherwise returns **false**.
- **fireEvent(Object msg)** – adds a message to the statechart queue.

To find out the current state of the statechart, use these methods:

- **boolean isStateActive(short state)** – returns **true** if a given state (simple or composite) is active, otherwise returns **false**.
- **short getActiveSimpleState()** – returns the currently active simple state.

You can ask the active object where the statechart belongs (not the statechart itself!), whether one state contains another:

- **boolean stateContainsState(short compstate, short simpstate)** – tests if **compstate** contains **simpstate** directly or at a deeper level.
- **short getContainerStateOf(short state)** – returns the immediate container of a given state, or -1 if this is a top-level state.

The statechart transitions expose these methods:

- **double getRest()** – returns the time remaining to the scheduled transition firing, or infinity if the firing is not scheduled.
- **boolean isActive()** – returns true if the transition firing is scheduled, and false otherwise.

During the code generation, the state names become constants of Java type short, and transitions – objects of one of the subclasses of **Transition**. They are all defined at the level of the active object; therefore, two states or transitions cannot have the same name even if they are in different statecharts. When referring to the states or transitions from outside the active object, you should prefix them with the active object name. For example, to test if a statechart **gearbox** of the active object **car** is in the state **Parked** you should write:

```
car.gearbox.isStateActive( car.Parked )
```

Such expressions are used, when, for example, you collect statistics on a collection of agents – active objects of the same class each having a statechart inside.

7.5. Viewing and debugging the statecharts at runtime

At runtime, AnyLogic highlights the active states of statecharts, the transitions that are scheduled to fire or have just fired. For active timeout and rate transitions, the remaining time to firing is displayed.

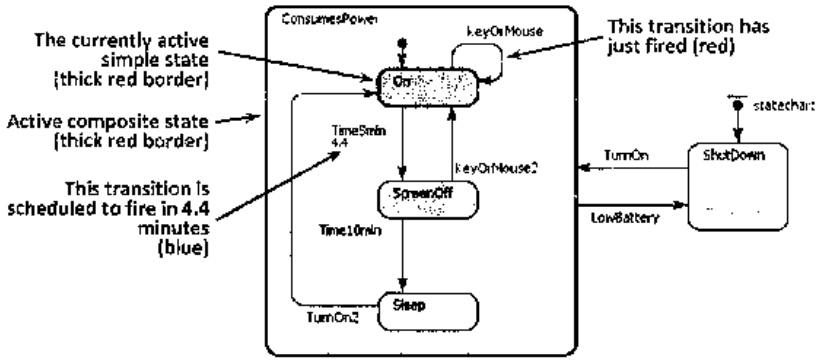


Figure 7.17 Statechart highlighting at runtime

Just like events, the scheduled statechart transitions can be observed in the **Events** view (see Chapter 8, "Discrete events and Event model object" to find out how to open the **Events** view). For example, for the statechart state shown in Figure 7.17 above the **Events** view will display transition **Time5min** that is scheduled and, in a

different list, the transition **LowBattery** that is constantly monitoring the battery condition.

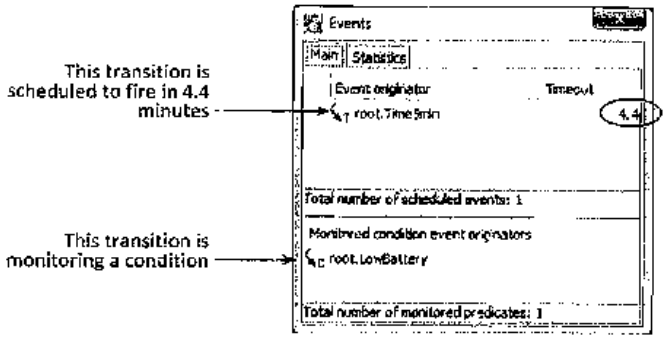


Figure 7.18 Events view displays the scheduled transitions and transitions that monitor conditions

To further debug the statecharts, you can use the step-by-step execution mode, the breakpoints, and also you can write to the model log (see Example 8.1: "Event writes to the model log every time unit") from statechart actions.

7.6. Statecharts for people's lives and behavior

Example 7.2: Life phases

The statechart for an individual's life phases described here is a simplified version of the one from the (Wallis, Paich & Borshchev, 2004). At the topmost level, the individual can be either **Alive**, or **Dead**, see the two highest level states. If the individual is alive, he can be in one of four different life phases: **Child**, **Adult**, **MidAge**, or **Senior**.

As in general, the individuals can appear in the model at different ages (e.g. as a result of in-migration), the statechart may initialize in either of the states, depending on the age. (Note that the statechart entry point points not to a state directly but to a decision point with four branches.) The transitions between the life phases are triggered by stochastic timeouts. For example, a transition from Adult to **MidAge** happens when the person is *around 49*, which is modeled by the timeout **normal(49,5) – age()**, where **normal(49,5)** is a normally distributed age with mean 49 and standard deviation 5, and **age()** is the age the person became **Adult** (remember that the timeout expressions are evaluated at the moment the statechart gets into the transition source state, in this case Adult).

The **Adult** life phase is further decomposed to describe family-related behavior. The decisions in this section of the statechart (to have family or not, how long to wait

before the first kid, how many kids to have, etc) are also stochastic and may depend on the gender, the level of education, the cultural norms, etc. If this statechart is inside an agent in an agent based model, the act of childbearing may result in a new agent added to the model, who may inherit the characteristics of the parents. The event of death then may delete the agent from the model.

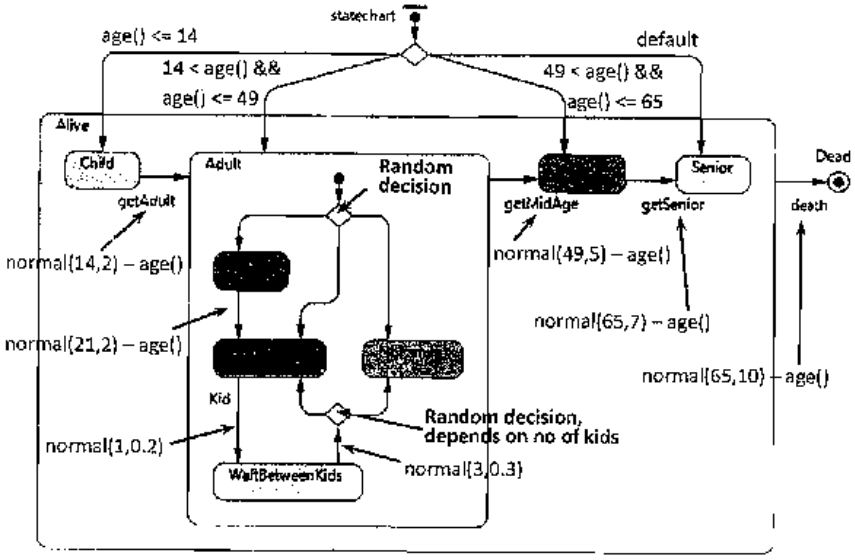


Figure 7.19 Statechart for life phases

A model of an individual may have other statecharts or other behavior "threads" that can communicate with the life phase statechart. For example, the education, the employment, the purchase behavior, etc. may depend on the life phase and may, in turn, affect the decisions made in the life phase statechart.

Example 7.3: Adoption and diffusion

The statechart in this example is a fundamental construct for all individual based (agent based) models where people are first being influenced to adopt an idea, a product, etc., and then, having adopted it, spread it by word of mouth. A very similar statechart is used in the models of diffusion of infectious diseases (see Example 7.4: "Disease diffusion").

We distinguish between two states of a person: **PotentialAdopter** and **Adopter**. People are sensitive to advertizing (by which we mean any non-personal information sources) and to word of mouth, hence the two transitions from the first state to the second. The **Advertizing** transition fires after a stochastic timeout, which models

random (and comparatively low) sales/adoptions caused by advertizing. **AdEfficiency** is the corresponding parameter. The **WordOfMouth** transition is triggered by a message received from another person. Having received the message "Good stuff!", the person will adopt the "stuff" with a certain probability – **AdoptionFraction** – otherwise he returns to the state **PotentialAdopter**.

Although any firing of **WordOfMouth** resets the previously scheduled **Advertizing** transition, we should not bother: the exponentially distributed timeout (which, in fact, is a rate transition) survives any number of resets without violation of distribution function.

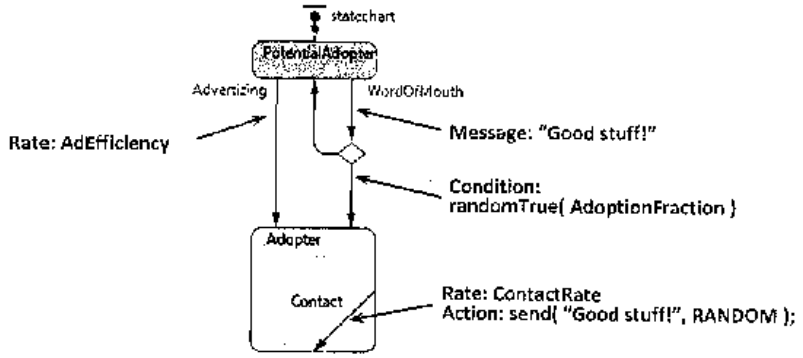


Figure 7.20 Statechart for adoption and diffusion

In the **Adopter** state, the person periodically contacts other people (randomly picked from the whole population, from personal contacts, neighbors, etc.) and tells them how good the stuff he has adopted is. This is modeled by an internal (see Section 7.3) **Contact** being executed in a loop at **ContactRate**.

Example 7.4: Disease diffusion

The statecharts used in disease diffusion models are similar to the one for product/idea adoption described in the previous section. We will consider a fairly general case. The disease only spreads from one person to another (therefore, as opposed to in the adoption model, there is no "advertising" channel). Having been infected, the person goes through several stages with different degrees of infectivity and different contact rates. Finally, having recovered from the disease, the person becomes immune (insensitive to infection).

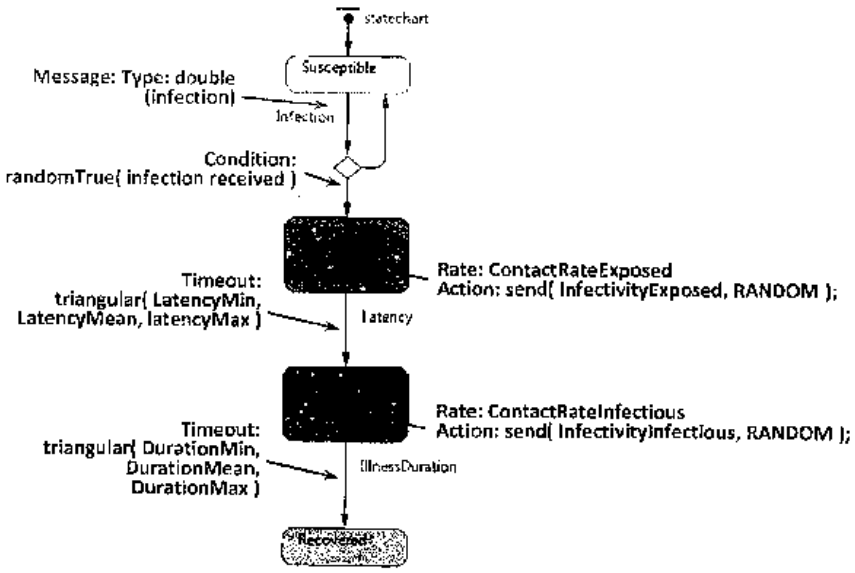


Figure 7.21 Statechart for agent based SEIR model

The statechart in Figure 7.21 can be used in an agent based SEIR (Susceptible Exposed Infectious Recovered) model of diffusion of an infectious disease. Initially, the person is **Susceptible** (can be infected by contacting other people). The infection passed during a contact can be of different degrees; therefore, a double value 0...1 is used as the infection message. Having received an infection, the person becomes **Exposed** with the corresponding probability, otherwise he remains **Susceptible**. The **Exposed** state corresponds to the disease latency period when the symptoms have not yet developed, so he continues contacting other people at a regular rate **ContactRateExposed** (see internal transition **ContactE**). During each contact, the person passes infection, although its doze is comparatively low: **InfectivityExposed**. After the latency period, the person becomes ill (state **Infectious**) and limits his contacts to the minimum (**ContactRateInfectious**). However, each contact results in a serious doze of infection being passed: **InfectivityInfectious**. Both latency period duration and illness duration are triangularly distributed. When the person recovers, he becomes immune and (as long as there is no transition from **Recovered** back to **Susceptible** in this statechart) immune forever.

Example 7.5: Purchase behavior with a choice of two competing products

This statechart is yet another extension of the generic statechart for adoption and diffusion (see Example 7.3: "Adoption and diffusion"). It can be used in agent based models of consumer markets. The additional features captured are:

- The competition between two companies (or between your company and all others).
- The limited usage time of the product and the need for repeated purchases.
- The potential limited availability of product (e.g., because of supply chain problems).
- Some sort of brand loyalty and switching behavior.

There are two competing products on the market: A and B. Initially, the consumer is in the **PotentialUser** state, where he has not yet decided to buy that kind of product at all. Either by advertizing or by of word of mouth, the consumer becomes convinced and wants to buy a product of a particular brand: states **WantA** and **WantB**. (This happens in the same way as in the generic adoption statechart, although the advertizing effectiveness may be different for different brands.) If the product of the corresponding brand is available (the stocks can be modeled by for example system dynamics stock variables **RetailerStockA** and **RetailerStockB**, by variables, or by discrete event elements like **Queue**), the consumer immediately proceeds to the **UserA** or **UserB** state, otherwise he waits. In this example, waiting for a particular brand is limited to 2 days. After that, the consumer gives up (his loyalty "expires"), and he is ready to buy any brand that is available (state **WantAnything**).

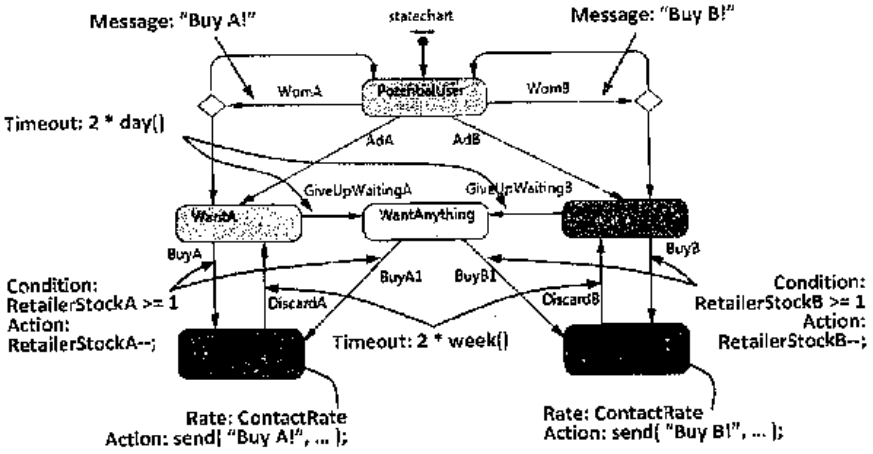


Figure 7.22 Statechart for consumer's choice of two competing products

The product's usage time is two weeks, and after that period the product is discarded and the consumer needs to buy another one, preferably of the same brand (transitions **DiscardA** and **DiscardB** bring the consumer back to **WantA** and **WantB** states).

? One of the assumptions in this model is that once the consumer has bought a particular brand, he stays with it and ignores the advertizing and word of mouth for the competing brand, so the only reason for switching is unavailability of the currently chosen brand. How would you modify the statechart to drop this assumption?

7.7. Statecharts for physical objects

Example 7.6: Generic resource with breakdowns and repairs

Sometimes, agent based models include objects that are used and shared by other objects (agents) and can be treated as resources (similarly to resources in process models). A very generic statechart for such agent-resource is shown in Figure 7.23. The resource can be either **Operational** or **OutOfOrder**. The failures occur sporadically, and the time between failures is frequently modeled as exponentially distributed with a certain known mean value (*MTBF* – Mean Time Between Failures); therefore, we can use a rate transition with the rate **1/MTBF**. Similarly, for the repair time we can use another rate transition with the rate **1/MTTR** (*MTTR* – Mean Time To Repair).

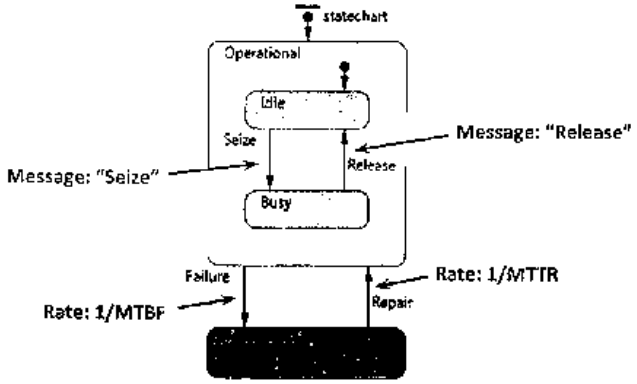


Figure 7.23 Statechart for generic resource

Seize and release operations can be modeled by, for example, message-triggered transitions toggling the statechart between **Idle** and **Busy** states.

? While the failure can occur both in Idle and Busy states, the Repair transition always brings the resource to the Idle state. Is there a simple way to restore the state?

Example 7.7: Delivery truck

Statecharts are extensively used to model operation of various physical objects: vehicles, cranes, elevators, machines, infrastructure elements, etc. Consider a delivery truck, which delivers goods to clients from a warehouse. If the truck is modeled as an autonomous object (agent), it makes a lot of sense to define its behavior in the form of a statechart like the one in Figure 7.24.

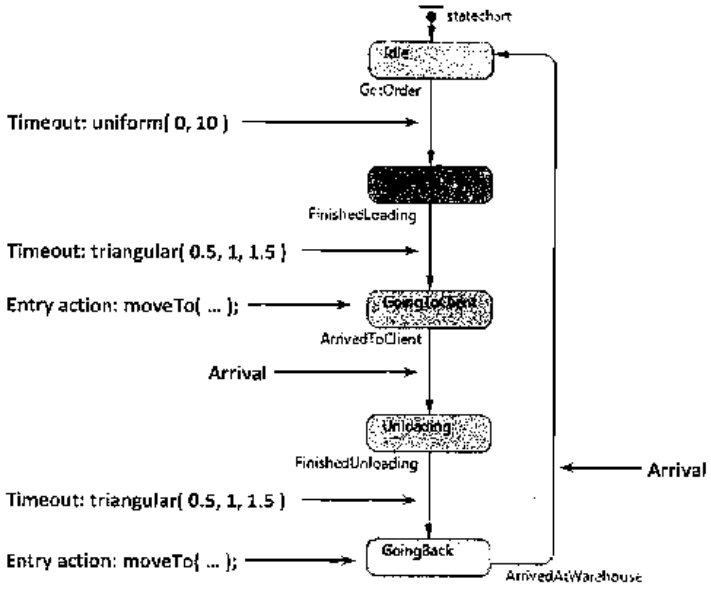


Figure 7.24 Statechart for delivery truck

Initially, the truck is in the **Idle** state (parked at the warehouse) and waits for a delivery order. Once the order is received, it starts loading, which takes 1 hour on average. Having been loaded, the truck departs to a client (see the call of agent's method **moveTo(...)** in the entry action of **GoingToClient** state). The transition **ArrivedToClient** is triggered by arrival (this type of trigger is only available in active object that are declared as agents). Unloading and returning to the warehouse is modeled in a similar way.

Example 7.8: Aircraft maintenance checks

In the agent based models of fleets of aircrafts, rail cars, trucks (or, in general, any pools of objects that are subject to periodic maintenance, such as houses, water or electricity delivery infrastructure, etc.) statecharts can be efficiently used to model the maintenance rules and the resulting availability of the objects. Consider an

aircraft. The periodic maintenance checks have to be done after a certain amount of time or usage. The most common checks are:

- **ACheck** – due every month or each 500 flight hours; done overnight at the airport gate.
- **BCheck** – due every 3 months; also done overnight at the gate.
- **CCheck** – due every 18 months or each 9000 flight hours; performed at a maintenance base or at a hangar and takes about 2 weeks.
- **DCheck** – due every 5 years; performed at a maintenance base and takes 2 months.

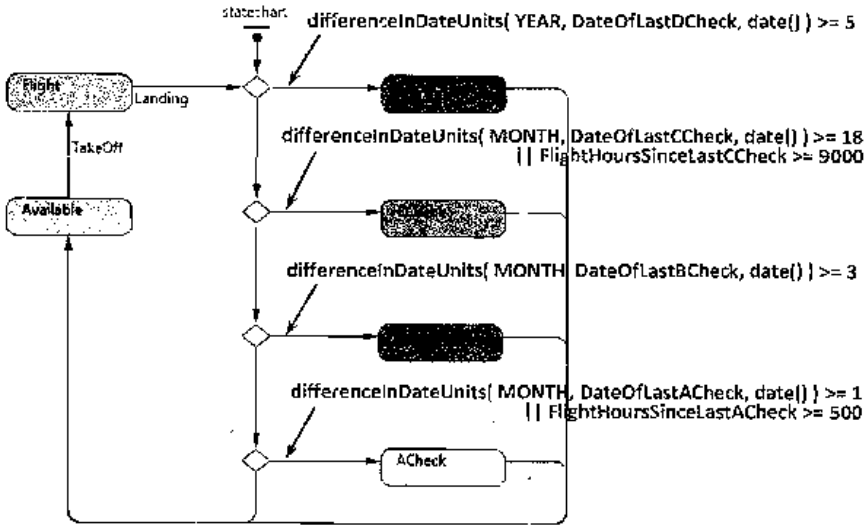


Figure 7.25 Statechart for aircraft maintenance checks

A possible statechart for aircraft maintenance is shown in Figure 7.25. When not undergoing maintenance, the aircraft is either **Available** (ready to fly), or in the **Flight** state. The transitions **TakeOff** and **Landing** can be triggered by timeouts or by actual commands, depending on how you wish to model the aircraft operation.

Upon each landing, the statechart determines if the aircraft is due for maintenance, starting from the most "heavy" DCheck. The condition in a branch to **DCheck** state is:

```
differenceInDateUnits( YEAR, DateOfLastDCheck, date() ) >= 5
```

Here, **date()** is the current model date, and **DateOfLastDCheck** is assumed to be a variable of type **Date**, where the date of last D Check is stored (this can be done in an exit action of the **DCheck** state). The function **differenceInDateUnits()** called with the

first parameter set to YEAR returns the number of years between the two given dates. If that condition is false, the statechart evaluates the next one – for CCheck:

```
differenceInDateUnits( MONTH, DateOfLastCCheck, date() ) >= 18 ||
FlightHoursSinceLastCCheck >= 9000
```

This condition has two parts: one based on time since the last CCheck and another – based on the flight hours since the last CCheck, which are stored in a variable **FlightHoursSinceLastCCheck** and are updated on each landing (remember that || is Java syntax for OR). The other two conditions for BCheck and ACheck are defined similarly. It is important that when a maintenance task is performed, it resets the dates and flight hours for itself and for all lighter maintenance types. The transitions outgoing the maintenance states represent maintenance times: **2 * week()**, **15 * hour**, etc.

? In the statechart, the maintenance conditions are only checked upon each landing; therefore, the aircraft will not miss the time-based check only if it flies regularly. If, however, there are significant idle periods, the need for time-based checks can potentially be detected too late. How would you modify the statechart to drop the assumption of flight regularity?

7.8. Statecharts for products and projects

Example 7.9: Product life cycle, including NPD

Statecharts can be used to model the lifecycle of the company's projects or products, (e.g. in the agent based models built to support portfolio management). A very generic statechart for a product lifecycle including NPD (New Product Development) phase is shown in Figure 7.26. At the top level, we distinguish between **NewProductDevelopment** and **InMarket** states. NPD in turn breaks down into four simple states ("New product development", n.d.) (these may be different for different types of products):

- **FrontEnd** – opportunity identification, idea generation and screening, business analysis.
- **Prototyping** – producing a physical prototype of the product
- **Testing** – testing the product prototype in typical usage situations, making adjustments.
- **Development** – planning and implementing engineering operations, quality management, supplier collaboration, etc.

The transitions between these states have the meaning of successful accomplishment of the corresponding phase and proceeding to the next one.

Technically, they can be stochastic timeout transitions where distribution of phase durations is based on expert knowledge, or may depend on other parts of the model (e.g. on resource availability). At any time during the NPD phase, the process can be suspended and then resumed, or killed. These are the company decisions, which may depend on many different factors.

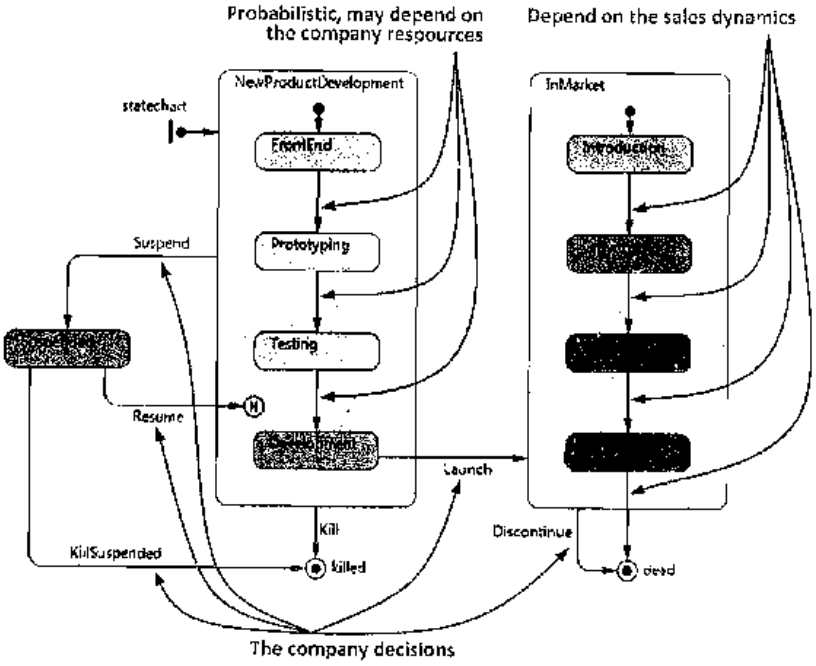


Figure 7.26 Statechart for product life cycle

When the development phase is successfully completed, the product is launched onto the market. The **InMarket** state, in turn, is broken down into another four states (QuickMBA.com, 2010):

- **Introduction** – sales are low, the company is building the product awareness, advertizing costs are high.
- **Growth** – strong growth in sales, little competition, the company is building a market share, promotion is aimed at a broader audience.
- **Maturity** – the strong growth in sales diminishes, competition may appear with similar products: the objective is to defend the market share.
- **Decline** – sales are declining because of market saturation, or the product becomes technologically obsolete: the marketing support is typically reduced.

Again, at any moment, the company is able to discontinue the product (although it is typically done at the decline phase). The transitions between these states typically depend on the sales and market dynamics, which are modeled outside of the statechart.

Example 7.10: Pharmaceutical NPD pipeline

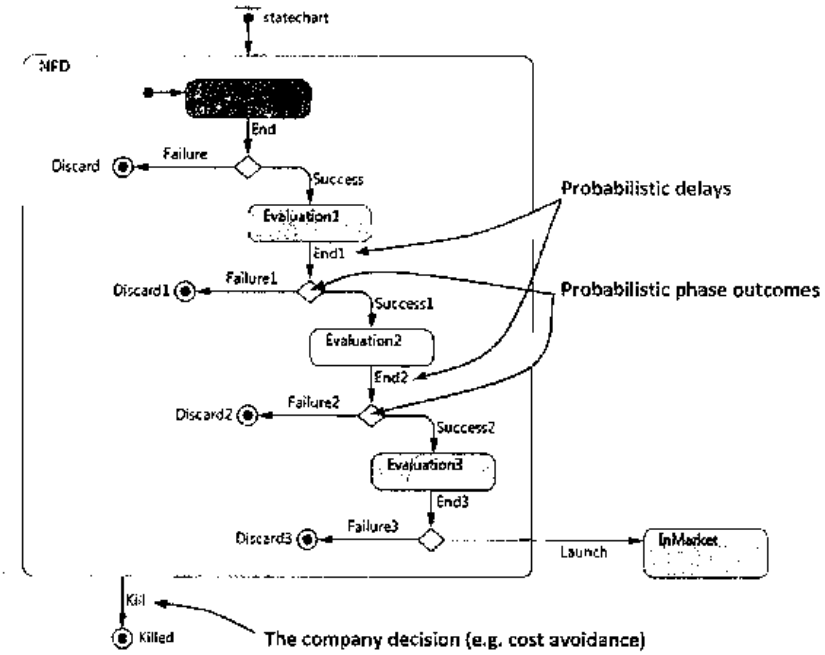


Figure 7.27 Statechart for pharmaceutical NPD pipeline

Depending on the industry, the product lifecycle structure may be slightly different from the generic one described in the previous example. For example, the pharmaceutical and biotechnological NPD pipelines require that the product is put through three clinical evaluation phases before it can be launched. (Solo & Paich, 2004) suggest using a statechart similar to the one in Figure 7.27.

The probability distributions used to model the durations of each phase (the timeout transitions: **End**, **End1**, etc.) are parameterized using the company's statistics and expert knowledge, as well as the probabilities of success and failure. You can also associate the resource consumption rate with each phase.

7.9. Statecharts for timing

Example 7.11: Statechart for shop working hours

States and time-driven transitions can be efficiently used to define sequences of time intervals, such as working hours, schedules, shifts, etc. Consider a shop in a southern country with these working hours:

- Weekdays: 9AM to 1PM, then 4PM to 8PM
- Saturday: 10AM to 2PM
- Sunday: closed

The statechart in Figure 7.28 defines the sequence of open and closed hours of the shop and takes daylight saving time into consideration.

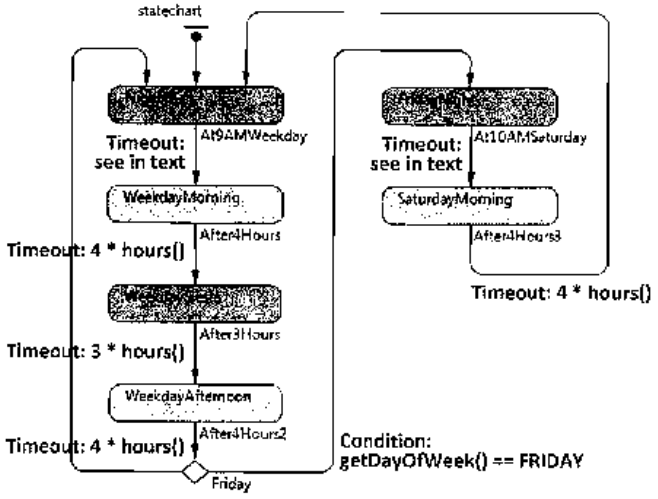


Figure 7.28 Statechart for shop working hours

The weekday sequence "open for 4 hours – closed for 3 hours – open for 4 hours" is implemented straightforwardly using three states and three timeout transitions. The only challenge is to initiate that sequence: we do not know when the statechart has entered the **Night** state, but the transition from **Night** to **WeekdayMorning** has to be taken each weekday at 9AM exactly. Therefore, the timeout in the transition **At9AMWeekday** is defined this way:

```
dateToTime( toDate( getYear(), getMonth(), getDayOfMonth(), 9, 0, 0 ) ) +
    toTimeout( DAY, getDayOfWeek() == SATURDAY ? 2 : 1 ) -
    time()
```

The expression (which is evaluated upon *entering* the **Night** state) has the following meaning: the function **toDate(getYear(), getMonth(), getDayOfMonth(), 9, 0, 0)** returns the calendar date/time corresponding to 9AM of the *current day*, i.e. of the day the shop was closed for the night. This date value is then converted to model time using the **dateToTime()** function. To obtain the model time corresponding to 9AM of the *next day*, we need to add either 1 day or, if today is Saturday, 2 days to skip Sunday. This is done using the expression **toTimeout(DAY, getDayOfWeek() == SATURDAY ? 2 : 1)**, which handles the daylight saving time (see Section 16.2) correctly. Then we subtract from 9AM of the next day the current time **time()** – and we get the amount of time we need to spend in the **Night** state. Similarly, we handle the time spent in the **FridayNight** state as follows:

```
dateToTime( toDate( getYear(), getMonth(), getDayOfMonth(), 10, 0, 0 ) ) +
   toTimeout( DAY, 1 ) -
   time()
```

Here we obtain 10AM on the current day (which is Friday) and add 1 day for Saturday.

Note that the statechart entry point points to the state **Night**; therefore, the model start time must be either between 2PM Saturday and 9AM Monday or between 8PM Monday to Thursday and 9AM of the next day.

Chapter 8. Discrete events and Event model object

8.1. Discrete events

The terminology

For the sake of clarity, we need to establish clear definitions for the terms used in this book to avoid confusion with other uses the reader may have come across. The terms *discrete event modeling* or *discrete event simulation* are commonly used for the modeling method that represents the system as a process, i.e. a sequence of operations being performed over entities such as customers, parts, documents, etc. These processes typically include delays, usage of resources, and waiting in queues. Each operation is modeled by its start event and end event, and no changes can take place in the model in between any two discrete events. The term discrete has been in general use for decades to distinguish this modeling method from *continuous time methods*, such as system dynamics (see Chapter 5).

With the emergence of agent based modeling (see Chapter 3) the term "discrete event modeling" in its traditional sense created confusion since in most agent based models actions are also associated with discrete events, but there may be no processes, entities, or resources. Therefore throughout this book we will be using the term *process modeling* for the modeling method where entities use resources and wait in queues, and the term discrete event for the more general idea of approximating the reality by instant changes at discrete time moments.

Discrete events: approximation of real world continuous processes

The dynamics of the world around us appear to be continuous: there are no instant changes – everything takes non-zero time, and there are no atomic changes – every change can be further divided into phases. For example, an airplane landing includes: descending, touching the ground, slowing down along the runway, and taxiing to the gate. An employee leaving a company must: look for a new job, send out a resume, get interviewed, get an offer, and so on. However, depending on your level of abstraction, "airplane lands" or "employee leaves" can be considered as instant events. Their component detail may not be relevant.

In discrete event modeling we only consider important moments in the system's lifetime, treat them as instantaneous and atomic *events*, and abstract away from anything that goes on between two contiguous events – see Figure 8.1. These are some examples of events:

- A customer enters the supermarket
- A truck arrives to the warehouse bay

- A project is finished
- A patient recovers from a disease
- Inventory level falls below the threshold
- The product price is reduced by 30%

Though virtually nonexistent in the natural world instantaneous atomic events can be observed in some artificial environments such as computer systems. For example a credit card approval by a bank server, arrival of an SMS message to a cell phone, submitting a web form, etc.

All dynamics (i.e. all changes) in a discrete event model, be they process models or agent based models, are associated with events. Events are *instantaneous* (the execution of an event takes zero time) and *atomic* (event execution cannot be interrupted by, combined, or interwoven with another event). Events have the ability to schedule other events; the event of airplane take-off may schedule the event of landing. Events may be simultaneous and be scheduled to occur at the same time. In that case events are *serialized* by the simulation engine, i.e. executed in some order.

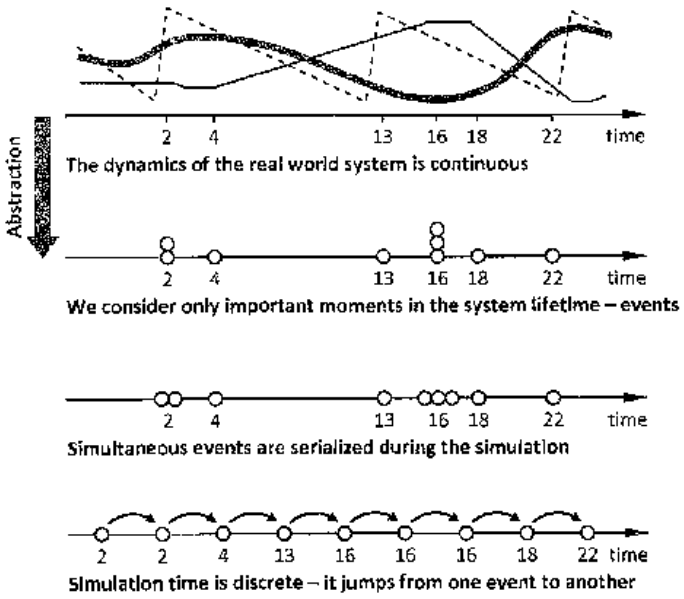

Abstraction
2 4 13 16 18 22 time
The dynamics of the real world system is continuous
2 4 13 16 18 22 time
We consider only important moments in the system lifetime – events
2 4 13 16 18 22 time
Simultaneous events are serialized during the simulation
2 2 4 13 16 16 16 18 22 time
Simulation time is discrete – it jumps from one event to another

Figure 8.1 Discrete event modeling and simulation

As long as nothing happens in between two subsequent events, the time in discrete event simulation is discrete: it jumps from one event to another.

Discrete event management inside AnyLogic engine

Consider how AnyLogic simulation engine executes a discrete event model. The engine maintains the *event queue* – the structure that contains all scheduled events. Look at the event queue example in Figure 8.2. Having executed the event a the engine advances the model clock to the group of simultaneous events {b,c,d}. Let the engine decides to execute b first (below we will consider how exactly the engine orders the simultaneous events). After b the engine chooses d and executes it (the model clock shows the same time). As a result of d two scheduled events get cancelled: c and f. As long as c is no longer the event queue, the clock jumps to e. e gets executed and schedules a new event i after the group {g,h}. The clock is advanced to {g,h}, and the engine goes on.

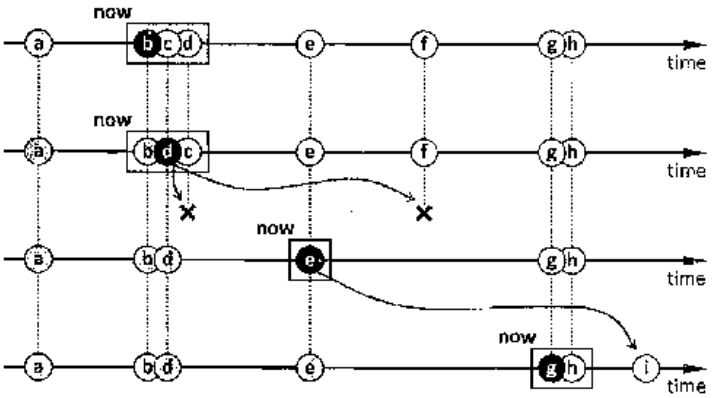


Figure 8.2 Event queue in AnyLogic simulation engine

There are two modes the simulation engine can serialize (impose order on) the simultaneous events. It can follow some deterministic internal order or choose events randomly. Executing an internal order is faster than a random order, but if your model is sensitive to the simultaneous event ordering the random mode will ensure you simulate a wider spectrum of possible scenarios. We suggest you build models to be insensitive to low-level event ordering. But in any case you can set up the serialization type at the experiment level:

▶ **To set the ordering mode for simultaneous events:**

1. Select the experiment and open the **Advanced** page of its properties.
2. Depending on what you want, check or uncheck the checkbox **Random selection mode for simultaneous events.**

8.2. Event – the simplest low level model object

In this section we will introduce the simplest object of AnyLogic modeling language called *event* (sometimes also called *static event* in contrast with dynamic event, see Section 8.3.

Please do not confuse *event objects* used by the modeler and *discrete events* in the simulation engine described in the previous section. A single event object can generate one or several discrete events during the simulation, which you can consider as its instances. Besides event objects, discrete events can be scheduled by statecharts (see Chapter 7).

The Event object provides a way to schedule a discrete event (see Section 8.1) or a sequence of discrete events directly into the simulation engine event queue. Theoretically, the event object is sufficient to build all kinds of discrete event models. In practice, however, modelers use higher level constructs such as the Enterprise Library objects or statecharts as well as fairly low-level events. Events are used mainly in the following cases:

Use case for events	Event type
Generate arrivals, births, etc in non-process models, e.g. in agent based models. (In process models you use the Source object for that)	Rate or cyclic timeout with stochastic recurrence time
Perform periodic actions, e.g.: daily patient review, annual budget planning, assignment of tasks at the beginning of the working day, scan the area every second, adjust direction every hour, replenish account every month, etc	Cyclic timeout with deterministic recurrence time
Perform sporadic actions: move to a new house on average every five years, change mind on average every year, change demand approximately every month, etc	Rate or cyclic timeout with stochastic recurrence time
Schedule delayed action "outside" the main action flow (which may be e.g. a process flowchart, or a statechart): the door closes in five seconds, the product is launched into the market in one month, the new train arrives with one minute interval, etc.	User-controlled timeout
Wait on a condition, e.g. inventory level reaching a threshold, or enemy aircraft is within the triggering zone	Condition
Periodically collect custom statistics, calculations, or write output	Cyclic timeout with deterministic recurrence time

Emulate external influence at particular time moments, e.g, demand increase, or commodity price change	Timeout in mode "occurs once"
Emulate user actions such as slowing the model down, zooming in, changing view, pausing, etc	Timeout in mode "occurs once"
Do something immediately, but in a different discrete event (because you wish to finish the current event first, for example)	User-controlled timeout with time = 0
Do something at time 0, but after the model is fully initialized and the startup code is executed	Timeout in mode "occurs once" with occurrence time = 0

▶ **To create an event:**

1. Open the **General** palette.
2. Drag the **Event** object from the palette to the canvas.

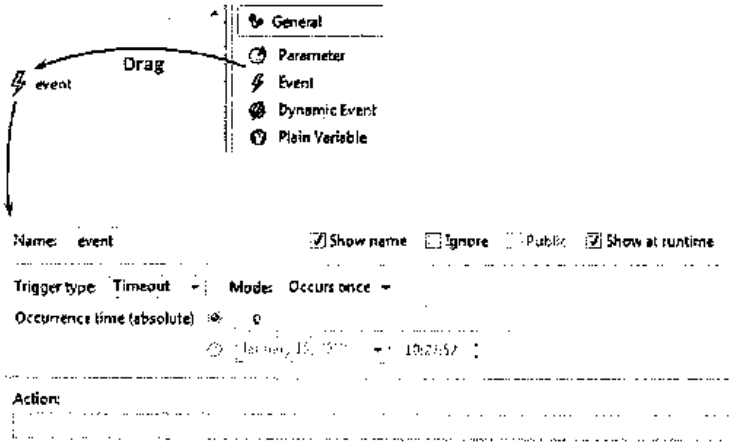


Figure 8.3 Creating a new event

All properties of the event are accessible on its **General** property page. The event has a **Trigger type** (**Timeout**, **Rate**, or **Condition**) and mode of operation. Event also has an **Action** field where you can specify what actually happens when the event occurs. Events may be of the following types (see also Figure 8.4):

- *Timeout event that occurs once* at the specified calendar time or model time
- *Cyclic timeout event* that occurs periodically with a certain recurrence time. You can also specify the time of the first occurrence
- *User-controlled timeout event* that occurs after its restart() method is called

- *Rate event* that occurs sporadically with exponentially distributed inter-occurrence times (Poisson stream of events)
- *Condition event* that occurs when a given condition becomes true

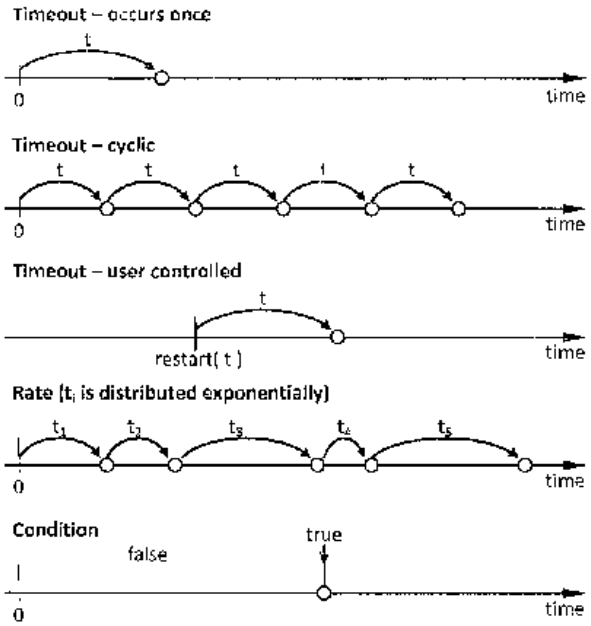


Figure 8.4 Event types (static events)

Below we will consider several examples of events.

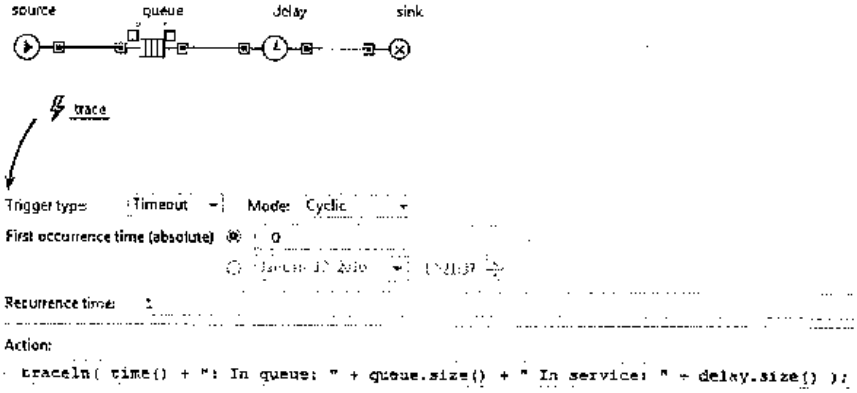
Example 8.1: Event writes to the model log every time unit

In the first examples we will create the simplest model of a queuing system and use a cyclic timeout event to write the status of the system to the model log. The system will consist of an entity generator, a queue, and a server.

▶ **Follow these steps:**

1. Press the **New** (model) button on the toolbar. In the **New model** wizard enter the model name and on the next page choose **Use template to create model** option and **Discrete Event** model template. Press **Next** and press **Finish** on the next page. A new process model is created and the editor of its **Main** object opens.
2. Run the model. Make sure the entities go through the system.
3. Go back to the editor of **Main** and open the **General** palette.

4. Drag the **Event** object from the palette and drop it anywhere below the flowchart. Set the name of the event to **trace**.
5. In the **General** page of the event properties set the **Mode** to **Cyclic** and leave the default value of the **Recurrence time** (**1**).
6. In the **Action** field of the event write:
   traceln(time() + ": In queue: " + queue.size() + " In service: " + delay.size());
7. Run the model and in parallel watch the **Console** window in the model development environment.



Run time – the model log

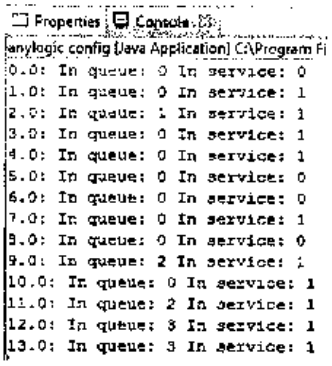

Properties
Console
anylogic config [Java Application] C:\Program Fi
0.0: In queue: 0 In service: 0
1.0: In queue: 0 In service: 1
2.0: In queue: 1 In service: 1
3.0: In queue: 0 In service: 1
4.0: In queue: 0 In service: 1
5.0: In queue: 0 In service: 0
6.0: In queue: 0 In service: 0
7.0: In queue: 0 In service: 1
8.0: In queue: 0 In service: 0
9.0: In queue: 2 In service: 1
10.0: In queue: 0 In service: 1
11.0: In queue: 2 In service: 1
12.0: In queue: 3 In service: 1
13.0: In queue: 3 In service: 1

Figure 8.5 Recurring timeout event is used to trace the model state

By setting the mode to **Cyclic** you make the event recurring and you can specify the recurrence time. The default value is **1**. The method **traceln()** writes to the model log, which you can view in the **Console** window of the model development environment (not in the model window). The argument of the **traceln()** function is a string. In this

example it is composed of string constants (like " In service: ") and numeric values returned by the method time() and the methods of the flowchart objects.

You can write arbitrary expressions in the **Recurrence time** field, not just a constant. The expression will be evaluated dynamically after each event occurrence. Therefore the inter-occurrence time can be made variable. In particular you can make it stochastic by using probability distribution functions in the expression.

Example 8.2: Event generates new agents

In the following example the event generates new agents; one agent every time unit on average. We will use the **New model** wizard to create an agent based model, but we will set the initial number of agents to 0.

▶ **Follow these steps:**

1. Press the **New** button on the toolbar. In the **New model** wizard enter the model name and on the next page choose **Use template to create model** option and **Agent Based** model template. Press **Next**.
2. In the next page of the wizard set the initial number of agents to **0**. Press **Finish**. A new agent based model is created and the editor of its **Main** object opens.
3. Run the model. Make sure there are no agents in the environment.
4. Go back to the editor of **Main** and open the **General** palette.
5. Drag the **Event** object from the palette and drop it anywhere below the environment object. Set the name of the event to **newAgent**.
6. In the **General** page of the event properties set the **Trigger type** to **Rate** and leave the default value of **Rate** (1).
7. In the **Action** field of the event write: **add_people();** (remember to use code completion available at Ctrl+Space, see Section 10.4).
8. Run the model. You should see agents appearing on each event occurrence.

The trigger type **Rate** = 1 means that the event will occur sporadically on average once per time unit, and the inter-occurrence times will be distributed exponentially. The method **add_people()** is generated automatically if there is a replicated embedded object with name **people** (this object was created by the **Agent Based** wizard).

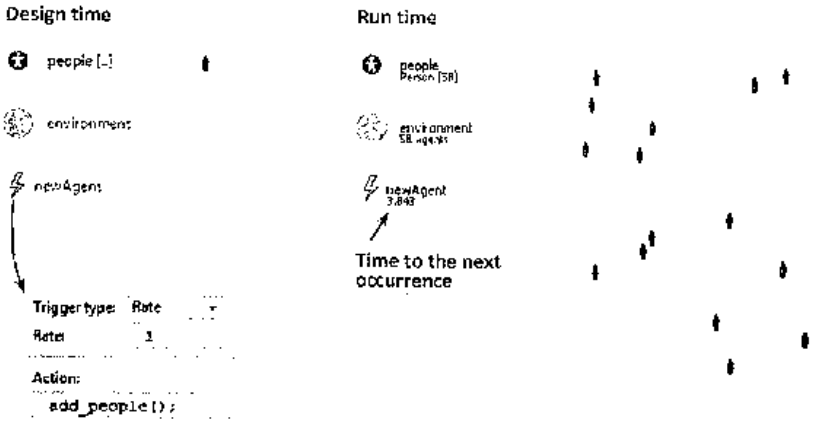


Figure 8.6 Event of Rate type dynamically adds new agents to the model

Events triggered by a condition

Event may be triggered by a *condition*. Condition is an arbitrary expression and may depend on the states of any objects in the whole model with continuous as well as discrete dynamics: system dynamic variables (see Section 5.1), statecharts (see Chapter 7), variables (see Section 10.3),Enterprise Library objects, etc.

The condition of such event is tested by the simulation engine when anything happens in the *same active object* (the same applies to the statechart transitions triggered by condition, see Section 7.3):

- Another event occurs
- A statechart transition is taken
- A parameter is changed
- A control of the active object has been accessed by the user
- If active object is an agent and it arrives to the destination point
- It active object is an agent and it receives a message

In addition, if the model contains any continuous dynamics (i.e. if there are any system dynamics variables in *any* of the active objects), the conditions of *all* active objects are tested on each integration step.

In most cases you should assume the condition is monitored "all the time" and the event is triggered as soon as it becomes true. However, if the condition depends on *other active objects*, it will not necessarily be tested when "remote" discrete changes take place there. Then, to avoid missing the exact moment of time when the

condition becomes true, you should explicitly call the method **onChange()** of the active object where the condition event is defined each time such change occurs.

After the event gets executed by the engine, it gets deactivated and stops monitoring its condition. This default behavior is useful because it prevents infinite event loops cases where the condition remains true after the event occurs. If you need the event to continue monitoring you should call its method **restart()** in the event action.

It is important to understand that some dynamics in the model that may seem to be continuous and is *animated as continuous* (such as movement of an agent, or of an entity in the network) is in fact modeled as two discrete events: one in the beginning and another one in the end. Therefore if you specify a condition like **agent.distanceTo(otherAgent) <= 50**, it may not be tested at the right time and you may miss the moment. To fix this you need to test the condition periodically by using e.g. a cyclic timeout event.

Example 8.3: Event waits on a stock reaching a certain level

In this example we will create a very simple system dynamics structure: a stock and a constant incoming flow filling the stock. The event triggered by condition will close the inflow when the stock reaches the level of 100 units. This example demonstrates one of the methods you can use to link continuous system dynamics model components with discrete ones.

▶ Follow these steps:

1. Open the **System Dynamics** palette and drag the **Stock** object to the canvas.
2. Drag the **Flow/Aux Variable** and drop it on the left of the stock. Change the name of the variable to **flow**.
3. Double-click the **flow** and drag the arrow to **stock**. This creates an inflow.
4. In the properties of **flow** check the checkbox **Constant.**. Then set the field **flow =** to **10**.
5. Open the **General** palette and drag the **Event** object to the canvas. Set the name of the event to **stockAt100**.
6. In the properties of the event set:
   Trigger type: Condition
   Condition: stock >= 100
   Action: flow = 0;
7. Run the model. Observe the value of the stock.

As long as we want to control the value of the **flow** from outside the system dynamics model, we need to mark it as a constant. This tells AnyLogic that the value

of the flow will not be continuously evaluated as a formula. The other important thing is that the condition of the event is not **stock == 100** but **stock >= 100**. The latter condition is correct because the value of the stock is integrated and grows in small discrete steps, therefore it is possible that it never gets exactly equal to 100, so that condition stock == 100 may never become true!

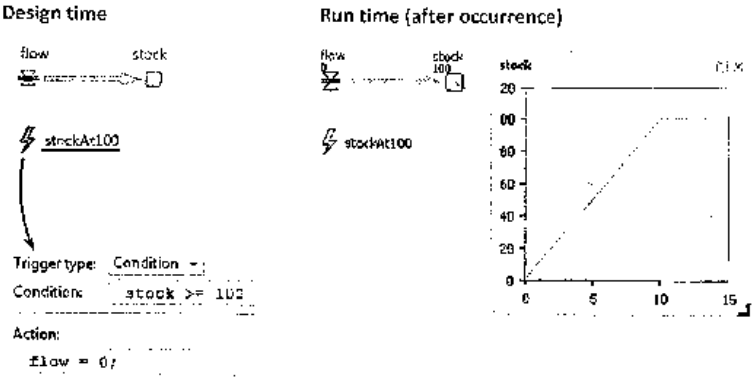


Figure 8.7 Event of Condition type triggered by a system dynamics variable

Please note that after the event is executed it stops monitoring the condition. This default behavior makes a lot of sense in this particular model where the condition remains true after the event occurrence. If it continued to monitor, it would be triggered infinitely in a loop not allowing the simulation time to progress.

An obvious question would be: but what if we want the event to be activated again when the stock falls below the 100 level? One of the solutions would be to create another event **stockBelow100** triggered by condition **stock < 100** and write in its action field **stockAt100.restart();**. At the same time you should add the line **stockBelow100.restart();** to the action field of **stockAt100**.

Example 8.4: Automatic shutdown after a period of inactivity

We will model automatic shutdown of a device, e.g. a laptop, done after a period of user inactivity. We will use two events: a rate event for sporadic user actions, and a manually controlled timeout event for the shutdown. Each user action will immediately wake up the laptop and reactivate the shutdown timeout.

▶ **Follow these steps:**

1. Open the **General** palette and drag the **Event** object to the canvas. Set the name of the event to **userAction**.
2. Drag another event from the palette and call it **shutdown**.

3. In the **General** property page of the **userAction** event set:
   Trigger type: Rate
   Rate: 0.1 / minute()
   Action: shutdown.restart();
4. In the **General** property page of the **shutdown** event leave the **Trigger type** default value **Timeout** and set:
   Mode: User Control
   Timeout: 10 * minute()
5. Open the **Presentation** palette, drag the **Oval** object and drop it anywhere beside the events. In the **General** page of the oval properties set the **Line color** of the oval to **null**.
6. In the **Dynamic** page of the oval properties set:
   Fill color: shutdown.isActive() ? limeGreen : gold
7. Run the model. Watch the times to the next occurrences of events and the color of the circle.

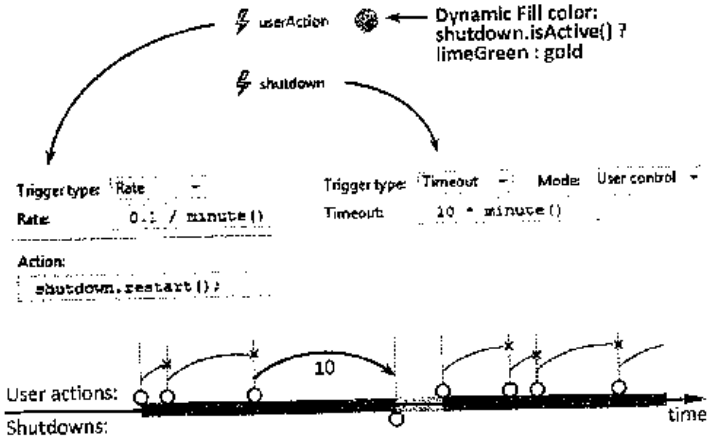


Figure 8.8 Rate event models user actions, and timeout event models automatic shutdown

The rate event **userAction** in this model occurs irregularly at exponentially-distributed time intervals on average 0.1 times per minute (i.e. once every 10 minutes). The **shutdown** event is a timeout of 10 minutes in the "user control" mode. It means that the timeout will not be activated until someone calls the **restart()** method. Every time the **userAction** event occurs, the timeout is restarted. If it happens before the previous timeout expires, the previous timeout is cancelled, and

a new one starts from the beginning. Therefore the **shutdown** event can only occur if the time between two subsequent user actions is greater than 10 minutes.

The functions like **minute()**, **hour()**, **day()**, etc. can be used in expressions to make them independent of time unit settings of the experiment.

The expression for the fill color of the circle tests whether the shutdown timeout is active and displays green or yellow color.

Example 8.5: Event slows down the simulation on a particular date

Suppose your model has some initialization or warm-up period that is not interesting to your audience, and you wish to skip it during the demonstration. You can use events to set up a demonstration scenario so that the model starts in very fast virtual time mode and, on a certain date, slows down to real time simulation mode.

▶ **Follow these steps:**

1. Press the **New** (model) button on the toolbar. In the **New model** wizard enter the model name and on the next page choose **Use template to create model** option and **Discrete Event Network Based** model template. Press **Next**.
2. In the next page of the wizard check the **Use resources** checkbox. Press **Finish**. A new model is created and the editor of its **Main** object opens.
3. In the **Projects** view select the **Simulation: Main** experiment of the model. The properties of the experiment show in the **Properties** view.
4. Open the **Model time** page of the properties and check the **Use calendar** checkbox. Set the **Start date** to January 1, 2010. Make sure that the model is not going to stop (the **Stop** is set to **Never**).
5. Open the **Window** page of the experiment properties. In the Show Statusbar sections part uncheck the **Model time**, **Simulation progress**, and **Memory** checkboxes and check **Model date** checkbox.
6. Open the **Presentation** page of the experiment properties and set the **Execution mode** to **Virtual time**.
7. Open the editor of the **Main** object where the flowchart is drawn.
8. Open the **General** palette and drag the **Event** object to the canvas. Set the name of the event to **slowDown**.
9. In the **General** page of the event properties leave the **Trigger type** set to **Timeout** and **Mode** set to **Occurs once**. Select the calendar entry for **Occurrence time**. Set the **Occurrence time** to February 1, 2010.
10. In the Action field of the event write: `getEngine().setRealTimeMode( true );`
11. Run the model. Watch how fast the first month is simulated. Then the model switches to the real time mode.

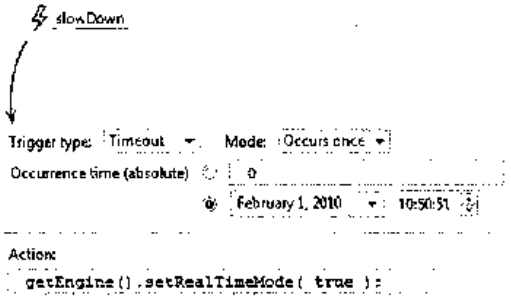


Figure 8.9 Event slows down the simulation on a particular date

In this example we are using calendar to define both the simulation start date and the event occurrence date (alternatively you could write `31*day()` in the **Occurrence time** field). The event **slowDown** occurs once and remains inactive after that. You can use such events to perform any one-time actions in the model.

Even if the event is specified with **Occurs once** option it can be scheduled again after occurrence by calling its `restart( time )` method.

Event API

Just like everything in AnyLogic, events are Java objects and expose their API to the modeler. Here is a summary of the methods:

- `reset()` – cancels the currently scheduled occurrence of the event, if any. In case the event is a cyclic timeout or a rate, the cycle would not resume until the `restart()` method is called.
- `restart()` – cancels the currently scheduled event, if any, and schedules the next occurrence according to the specified mode.
- `restart( double t )` – cancels the currently scheduled occurrence of the event, if any, and schedules the new occurrence in time `t`. If the event is a cyclic timeout or a rate, it will then continue occurring at the original timeout/rate.
- `suspend()` – cancels the currently scheduled occurrence of the event, if any, and remembers the remaining time to that occurrence so that it can be resumed by calling `resume()`. If the event is not scheduled at the time of calling `suspend()`, the subsequent resume will result in nothing.
- `resume()` – re-schedules the previously suspended event in the remaining time.
- `double getRest()` – returns the time remaining to the next scheduled occurrence of the event, or `infinity` if the event is not scheduled.

- **boolean isActive()** - returns **true** if the event is currently scheduled, and **false** otherwise.

8.3. Dynamic events

Imagine you are modeling product delivery by the postal service. After you send the product it gets delivered in 2 to 5 days and you ship several products per day, so multiple products may be in the delivery stage simultaneously. Which AnyLogic construct would you use? If delivery is a part of the process model you can use a **Delay** object of the Enterprise Library. However if there is no process (e.g. if this is an agent based model) you can use a lower level (and a more lightweight) construct – *dynamic event*. Dynamic event allows you to schedule an arbitrary number of occurrences in parallel, and each occurrence can be parameterized.

▶ **To create a dynamic event:**

1. Open the **General** palette.
2. Drag the **Dynamic Event** object from the palette to the canvas.

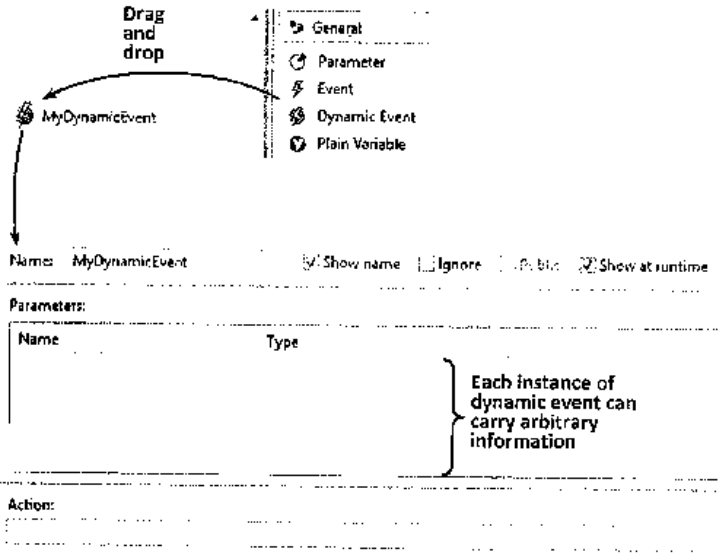


Figure 8.10 Creating a new dynamic event

Just like the static event described in the previous section, the dynamic event has an action field where you can define what should be done when the event occurs. However, unlike the static event, which persists throughout the simulation run, the dynamic event can be considered as a template, or a class, whose instances are

created when you schedule an event and are destroyed immediately after occurrence. An instance of a dynamic event only exists while it is scheduled. Each instance can carry arbitrary information, and that information can be accessed when the action is executed upon event occurrence. For example, if dynamic events are used to model product delivery, each event instance can carry the product being delivered.

To schedule a dynamic event you need to call an auto-generated method of the following syntax:

```
create_<Dynamic event name>( <time interval>, <parameter1>, ... )
```

The first argument in the method call is the time in which you want the event to occur. It is followed by the parameter values – one for each parameter defined in the event properties. The time interval can be deterministic as well as stochastic.

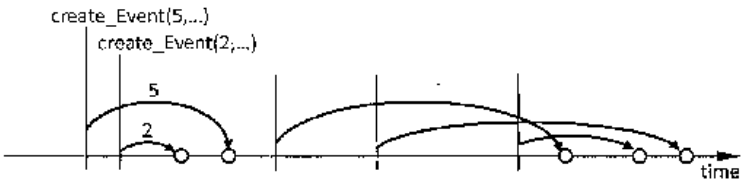


Figure 8.11 Dynamic events – multiple instances can be scheduled in parallel

Since the dynamic event that you create in the editor is actually a *class* of events, its name is capitalized according to Java writing convention.

Example 8.6: Product delivery

Let's model product delivery. The product in this simple model will be identified by a string. The delivery time will have a triangular distribution (see Section 15.1) with a minimum of 2, a maximum of 5, and a mode of 3 days. The shipment will be invoked by the user pressing a button, and when the product is delivered we will write a confirmation line into the model log.

▶ **Follow these steps:**

1. Open the **General** palette and drag the **Dynamic event** object to the canvas. Set the name of the dynamic event to **Delivery**.
2. In the **General** page of the dynamic event properties add a parameter **product** of type **String**.
3. In the **Action** field write: `traceln( time() + " Delivered " + product + "!" );`
4. Open the **Controls** palette and drag the **Button** object. Drop nearby the dynamic event.
5. In the **General** page of the button properties set the **Label** to **Ship**.

6. In the **Action** field of the button properties write:

```
traceln( time() + " shipping A..." );
create_Delivery( triangular(2,3,5), "A" );
```

7. Run the model. Try pressing the button multiple times. Notice the model log in the **Console** window of the model development environment.

The dynamic event **Delivery** has one parameter of type **String**. It means that each instance of **Delivery** will have a string attached to it. When the user presses the button, a new instance of the dynamic event is scheduled with the string "A" attached to it.

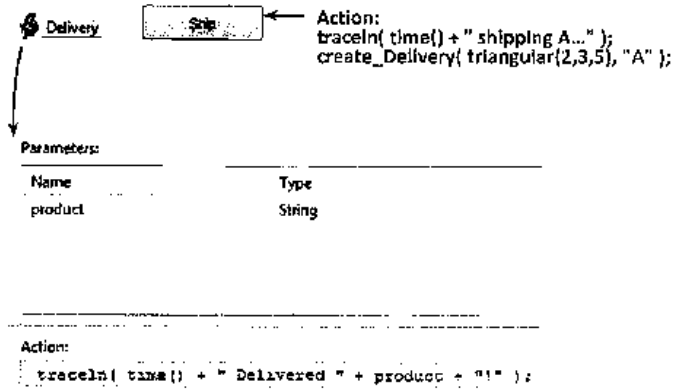


Figure 8.12 Dynamic event models delivery of a product

Let's extend the example a little. Instead of always shipping just one product "A" we will provide the user the ability to specify the product type in the edit box. We will also observe the scheduled dynamic events in the events view.

▶ **Add a text field for the product name**

8. Open the **Controls** palette and drag the **Edit box** item under the button as shown in Figure 8.13.
9. Select the button and change its **Action** to:

```
String product = editbox.getText();
traceln( time() + " shipping " + product );
create_Delivery( triangular(2,3,5), product );
```

10. Run the model.
11. Type something in the edit box and press the button. Watch the model log.
12. On the toolbar of the model window press the ▷ (**Customize toolbar**) button and choose **Events View** from the drop down menu. The **Events view** window should open on the right of the model window.

13. Try pressing the button multiple times and watch the events view.

The product type is now taken from the edit box, whose method **getText()** returns a String. The Events view displays all events (not just dynamic) currently scheduled in AnyLogic simulation engine. Each event is identified by the object that originated the event. **root.Delivery** means the root active object of the model (the instance of **Main**) and **Delivery** event within that active object.

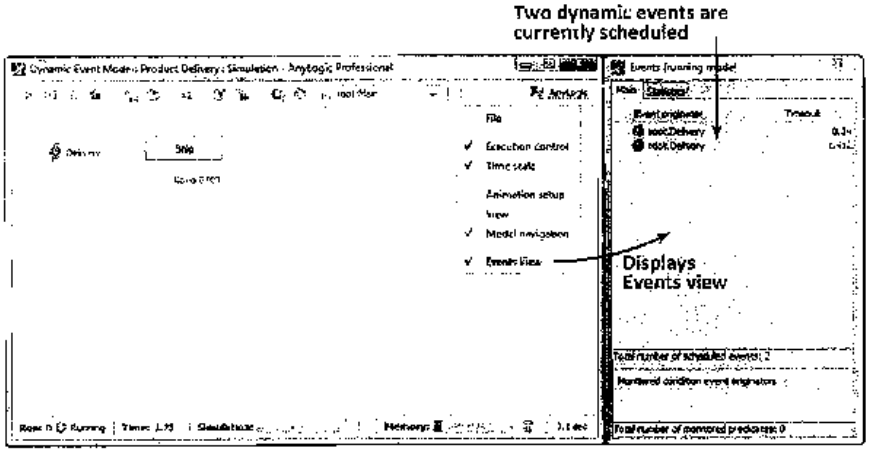


Figure 8.13 Scheduled dynamic events can be seen in the Events view

API related to dynamic events

Dynamic event items that you drag from the palette and drop on the canvas become a Java *class* (an inner class of the active object class), and each call of **create_<Dynamic event name>()** creates a new object – an *instance* of that class. Each instance of a dynamic event has the following API:

- **reset()** – cancels and destroys the instance of the dynamic event.
- **double getRest()** – returns the time remaining to the occurrence of the event.

To call the methods of a dynamic event instance you need to remember the instance. You can do it when you create it:

```
Delivery deliveryA = create_Delivery( 25, "A" );
...
double remainingTime = deliveryA.getRest();
```

You can retrieve the list of all dynamic event instances scheduled in an active object by calling the **ActiveObject**'s method

```
Set<DynamicEvent> getDynamicEvents()
```

The method returns a set of all dynamic event instances of all classes defined within this active object. To find out the class of an event you can use the instanceof operator, see Section 10.2, "Classes".

Chapter 9. Rails and trains

The AnyLogic *Rail Library* allows you to efficiently model, simulate and visualize any kind of rail transportation of any complexity and scale. Classification yards, rail yards of large plants, railway stations, rail car repair yards, subways, airport shuttle trains, rail in container terminals, trams, or even rail transportation in a coal mine can be easily yet accurately modeled with this library.

The Rail library integrates well with AnyLogic Enterprise and Pedestrian Libraries. This means you can readily combine rail models with the types of models supported by these libraries, namely: auto transport, cranes, ships, passenger flows, warehouses, manufacturing or business processes, and so on.

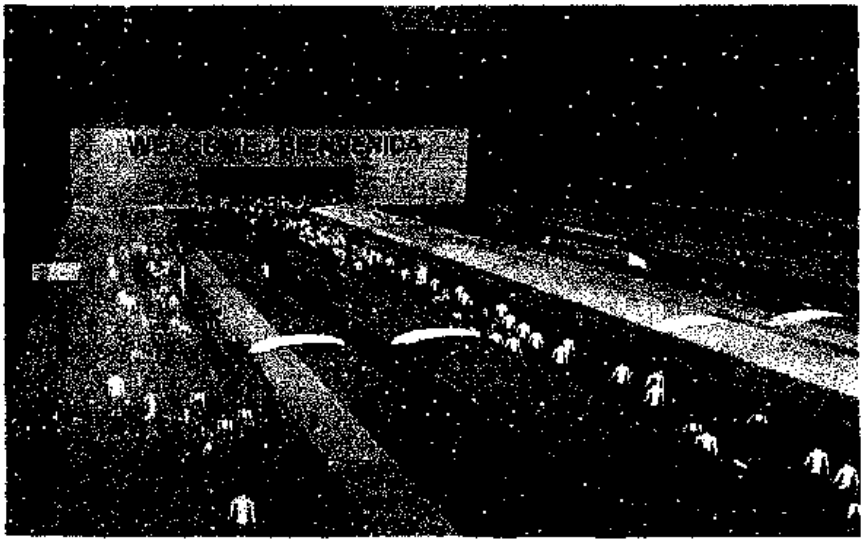


Figure 9.1 A Railway station model – Rail Library works together with Pedestrian Library

While the library supports detailed and accurate modeling (dimensions of individual cars, exact topology of tracks and switches, accelerations and decelerations of trains), the simulations it produces are very high performance. This is particularly important when you use the optimizer to identify the best management policies or most efficient operations.

The Rail Library supports 2D and 3D animation of tracks, switches, and rail cars. The **3D Objects** palette contains ready-to-use 3D objects for locomotives, several types of freight cars, and passenger cars. Since the Enterprise Library and the

Pedestrian Library also support 3D animation, you can now easily create full dynamic 3D models of subway and railway stations, airport shuttles, or any other system where rail transportation is combined with pedestrian flow (see Figure 9.1).

The two main components of a rail model are the rail topology and the operation logic. We will now discuss these in turn.

9.1. Defining the rail topology

The rail topology is specified by a group of shapes representing rail *tracks* and *switches*. Tracks are defined by polylines (**ShapePolyLine**), while switches are defined by circles (**ShapeOval**). Those shapes can either be drawn manually using the AnyLogic graphical editor or created programmatically, for example, by reading the layout from a database or a file (see Example 12.11: "Read graphics from a text file").

In a well-defined rail layout:

1. There are no shapes in the group other than polylines (**ShapePolyLine**) and circles (**ShapeOval**)
2. Each circle (switch) must contain exactly three polyline end points (tracks ends) within 2 pixels from the circle center (see Figure 9.2)
3. At each switch there must be at least two obtuse angles out of three between the track ends. The switch determines the routes based on those angles.

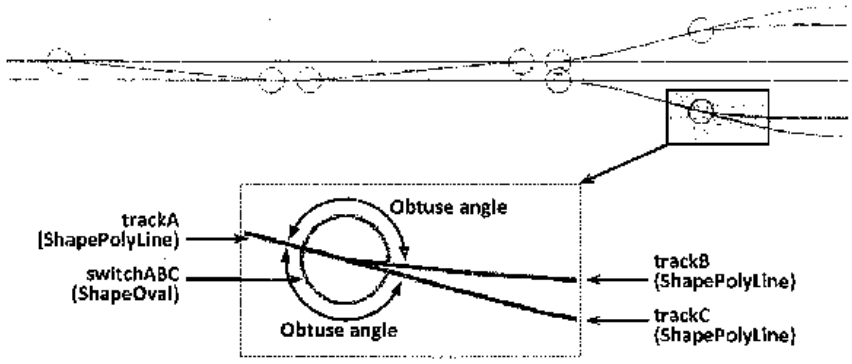


Figure 9.2 A fragment of rail topology near a railway station. Defining a switch

Train collisions at switches are detected automatically and errors are signaled. But note that the Rail Library will not automatically detect track crossings (i.e. places where two tracks cross each other without a switch) and it is the user's responsibility to make sure there are no train collisions in such places.

The characteristics of the track polylines and circles you use at design time (such as color, line width, or circle size) do not matter; at runtime they will be modified according to the scale settings and the color scheme defined in the **RailYard** object.

The curved track segments should be approximated with multiple point polylines. If a rail yard is drawn manually it may be convenient to have a CAD drawing or an image of the yard as a background, lock it (see Section 12.1), and draw the polylines above the image.

Having drawn the tracks and switches, you should group them, pass the group to the parameter **Rail yard shapes** of the **RailYard** object, and specify the scale in which you have drawn the yard (pixels per meter).

Example 9.1: A very simple rail yard

Let us draw a very simple rail configuration such as a main track and a branch line that may be used to wait while another train passes by.

▶ **Draw a rough sketch of the tracks:**

1. Open the **Presentation** palette. Double-click the **Polyline** item and draw a horizontal polyline with one segment of length 300 (click at the start point and double click at the end point). Call it **trackA**.
2. Ctrl+drag the polyline to create a copy. Place the left end of the second polyline at the right end of the first one. Call the new polyline **trackB**.
3. Extend **trackB** to the right so its length is approximately 500 pixels.
4. Ctrl+drag **trackA** again to create another copy of it and place it to the right of **trackB** as its continuation. Call the third polyline **trackC**. Now you have a straight line of length 300+500+300 = 800 consisting of three polylines.
5. Draw a small circle (with radius, say, 10) with the center at point where **trackA** connects with **trackB** (you may need to select **trackA** to see where it ends). Call the circle **switchABD**. Set the **Fill color** of the circle to **No fill**.
6. Create a copy of that circle and place it where trackB connects with trackC. Call the second circle **switchCBD**.
7. Draw the fourth polyline starting from the center of **switchABD** and ending at the center of **switchCBD**. Let this polyline initially have three segments as shown at the top of Figure 9.3. Call it **trackD**.

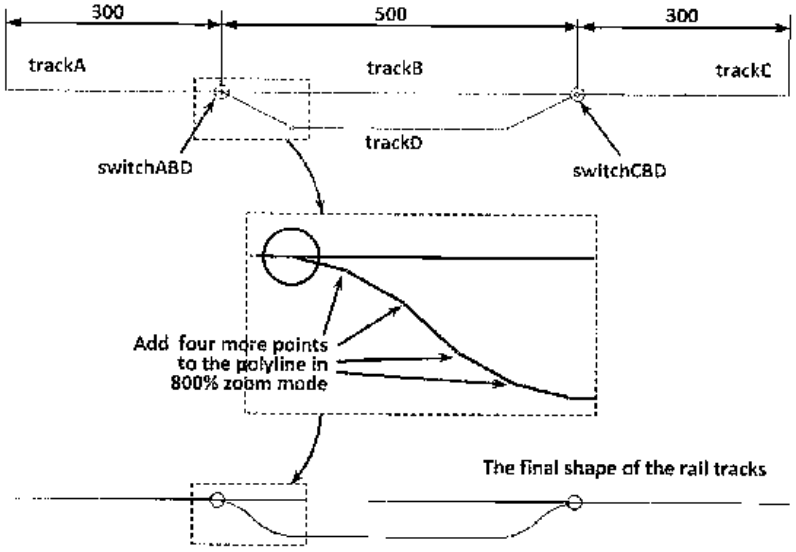


Figure 9.3 Drawing a very simple rail yard

Now let's make the shape of **trackD** more round (closer to what it would be in reality) by adding more points to the polyline.

▶ **Adjust the shape of the trackD to make it more realistic**

8. Double-click the header of the graphical editor window (it should read **Main**) to extend the editor to a full screen view.
9. Place the mouse cursor at **switchABD**, hold the Ctrl key and rotate the mouse wheel until the picture is zoomed to 800%. You will see the pixel-wise grid and will be able to move the polyline points by one pixel step.
10. Double-click **trackD** to add a point and adjust the point position. Add four points as shown in the middle of Figure 9.3.
11. Adjust the shape of the right end of **trackD** near **switchCBD**.
12. Return to 100% zoom and double-click the Window header again to restore the original windows layout.

Now that you have finished the layout drawing, you should group all those shapes, drag the **RailYard** object to the canvas, and tell it which group contains the rail yard shapes. We will assume the scale of the drawing is 2 pixels per meter.

▶ **Group all shapes and add the RailYard object:**

13. Select all the polylines and circles you have drawn.

14. Right-click the selection and choose **Grouping | Create a group** from the context menu. A group is created. Change the name of the group to groupRailYard.
15. Open the **Rail Library** palette and drag the **RailYard** object to the canvas.
16. In the General page of railYard properties type groupRailYard in the **Rail yard shapes (group)** parameter. Remember to use code completion.
17. Set the **Scale, pixels per meter** parameter to 2.
18. Run the model.
19. Zoom in the animation of the rail yard. Click on a switch to change its state.

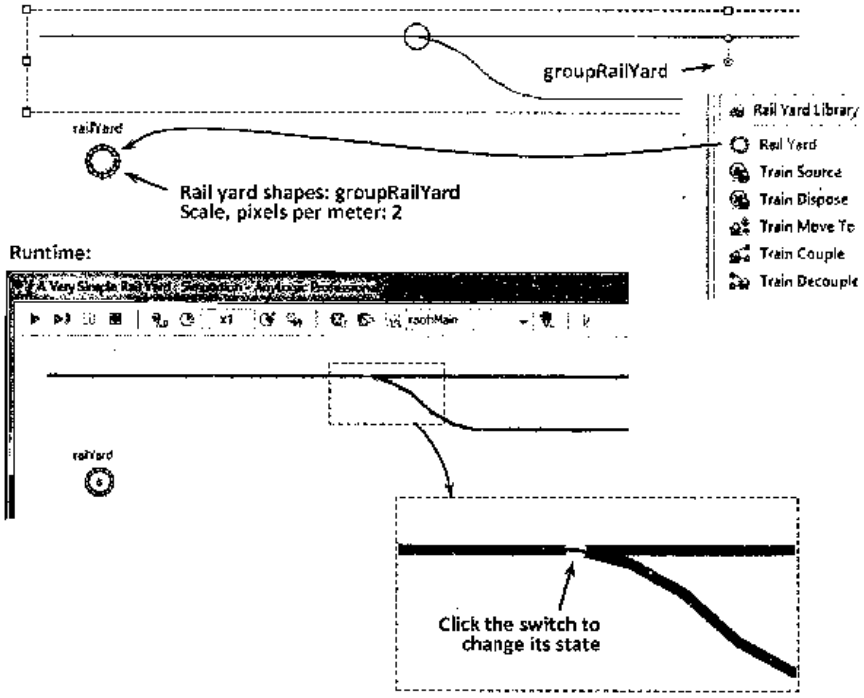


Figure 9.4 Adding the RailYard object. Animation of tracks and switches

Notice that the width and color of the tracks as well as the size and color of the switch circles at runtime are modified according to the settings in the bottom section of **RailYard** object parameters. (As mentioned above, these override the values you may have used in design.)

Now the rail yard is fully defined and ready to be used. We will illustrate how to let the trains into the yard later, in Section 9.2, "Defining the operation logic of the rail model".

3D animation of rail yards

The next thing is to enable 3D animation of the yard. This is done simply by making the group of shapes a 3D group (selecting its **Show in 3D scene** property). There is one thing you should check, though. If the polylines and switches were originally two-dimensional, their Z-height will automatically be set to 10 pixels, which may be undesirable. You should then modify the Z-height.

▶ **To enable 3D animation of the rail yard:**

1. Click the **groupRailYard** and select **Show in 3D scene** on the **General** page of its properties.
2. Open the **3D** palette and drag the **3D Window** to the canvas below the drawing. Change its size to, say, 1100x250 pixels.
3. Run the model. The tracks and switches have significant Z-height, which we need to fix.
4. Return to the editor. Right-click the group and choose **Select group contents** from the context menu. All rail yard shapes get selected.
5. In the **Advanced** property page of the multiple selection set **Z-Height** to 1.
6. Run the model again and view the 3D picture of the yard.

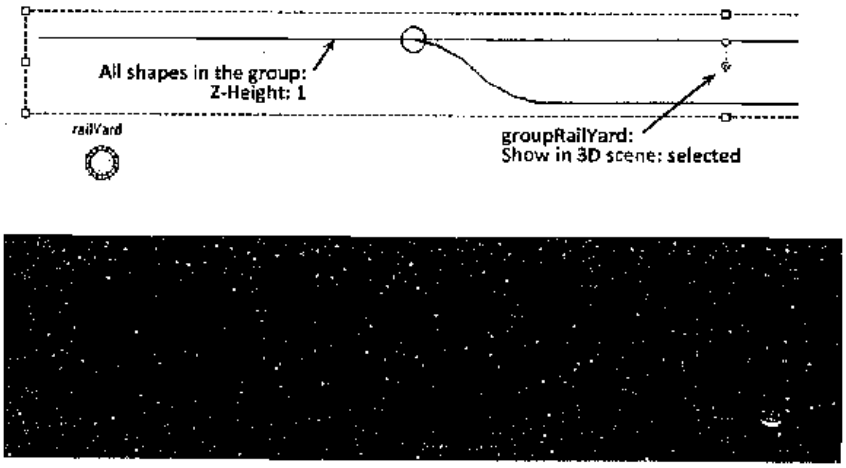


Figure 9.5 3D animation of the simple rail yard

Creating rail yards programmatically

For large rail yards it may make sense to read the layout of the yard from an external source and create tracks and switches programmatically instead of drawing them manually.

Example 9.2: Creating a rail yard by code

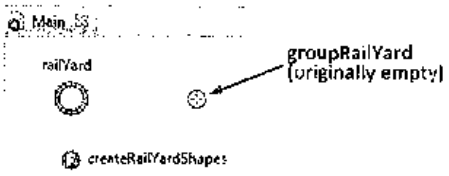
We will assume you have read the yard configuration from an external source and know the coordinates of switches and of all points of tracks (remember that curved tracks should be approximated by polylines). In this example we will create the polylines and circles using AnyLogic API, add them to a group and then dynamically set the group to the **Rail yard shapes** parameter of the **RailYard** object.

▶ **Follow these steps:**

1. Open the **Rail Library** palette and drag the **RailYard** object to the canvas at (50, 50). Do not change any of its parameters.
2. Open the **Presentation** palette and drag the **Group** object to, say, (150, 50). Call the group **groupRailYard**.
3. Open the **General** palette and drag the **Function** object to the canvas near the rail yard object. Call the function **createRailYardShapes**.
4. Enter the following code in the **Code** page of the function properties:

```
//create track A
ShapePolyLine trackA = new ShapePolyLine();
trackA.setNPoints( 2 );
trackA.setPoint( 1, 200, 0 );
trackA.setPos( 0, 0 );
groupRailYard.add( trackA );

// create track B
ShapePolyLine trackB = new ShapePolyLine();
trackB.setNPoints( 2 );
trackB.setPoint( 1, 400, 0 );
trackB.setPos( 200, 0 );
groupRailYard.add( trackB );

// create track C
ShapePolyLine trackC = new ShapePolyLine();
trackC.setNPoints( 3 );
trackC.setPoint( 1, 100, 50 );
trackC.setPoint( 2, 400, 50 );
trackC.setPos( 200, 0 );
groupRailYard.add( trackC );

//create switch at the end of trackA
ShapeOval switchABC = new ShapeOval();
switchABC.setRadius( 10 );
switchABC.setPos( 200, 0 );
groupRailYard.add( switchABC );

//refresh the group parameter of the RailYard
railYard.set_railYardShapes( null ); //just in case there were other shapes before
railYard.set_railYardShapes( groupRailYard );
```

5. Click anywhere in the canvas to display the active object properties. On the General page type the following in the **Startup code** field:
   createRailYardShapes();
6. Run the model. The three tracks and a switch appear at (150,50).



Runtime:

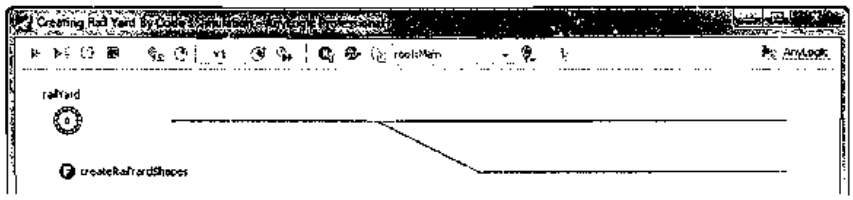


Figure 9.6 A rail yard created by code

In this model, we did not initially provide a group to the **RailYard** object, so the creation of tracks and switches was delayed. On startup, the function **createRailYardShapes** executes. It creates all shapes, adds them to the group, and then sets the group to the **RailYard**. When the **railYardShapes** parameter is changed, **RailYard** (re)builds the yard.

Please note that the position of the shapes is relative to the group where they are included. The polyline point coordinates are relative to the start point of the polyline, therefore 0 point always has coordinates (0,0).

Java class Track

For each polyline in the group of rail yard shapes (in other words, for each continuous track of the yard with no switches in the middle), the **RailYard** object creates a Java object of class **Track**. You will not typically need to use the objects of class **Track** and their methods: most references to tracks in the Rail Library process flowcharts are done by the polyline names; in addition, the **RailYard** object has a number of functions like **isTrackEmpty()**, or **getCarOnTrack()**, which also accept polyline names. However, in some cases the **Track** API might be useful.

To obtain the **Track** object that corresponds to a particular polyline you should call the function **getTrack()** of the **RailYard** object, for example:

```
Track track123 = railYard.getTrack( polyline123 );
```

The track has a start point (the point 0 of the polyline) and an end point, and therefore has an orientation. The exact position on the track is defined by the distance from the start point of the track, often referred as *offset*.

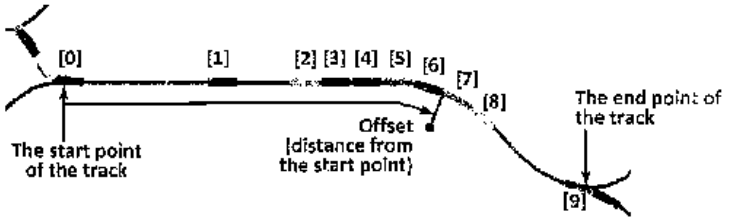


Figure 9.7 A track

The track knows about the presence of switches at the either end, if there are any. If there is no switch at one of the sides (an *open-ended track*), and a rail car exits the track at that side, the car leaves the rail yard model.

The track knows all cars that are (fully or partially) located on it, and you can obtain those cars using the **RailYard** object or **Track** object API. The track in Figure 9.7 will say that it has 10 cars on it, and will return them by index as shown.

At runtime, the color of an empty track is (optionally) different from the color of the non-empty track. Colors are set up at the bottom section of **RailYard** parameters.

Here are some methods of class **Track** (see *Library Reference Guides: Rail Library* (The AnyLogic Company, 2013) for the full list):

- **boolean isEmpty()** – tests if the track is empty, i.e. there are no cars that are (even partially) on the track. Returns true if the track is empty, false otherwise
- **int getNCars()** – returns the number of cars on the track (including those that are only partially on this track)
- **RailCar getCar(int index)** – returns a car on the track at a given position (**index**) counted from the beginning of the track, or null if there is no such car. All cars count: moving, standing, coupled, and cars that are only partially on this track.
- **double getFreeSpace(boolean fromstart)** – tests the availability of space on the track. If there are no cars on the track, returns **Infinity**. If there are cars, returns the distance from the track start or end point (depending on the parameter **fromstart**) to the nearest car. If there is a car partially entered or exited the track at a given side, returns a negative value.

- **Switch getSwitch(boolean atend)** – returns the switch (**Switch** object, not the circle shape) at the beginning or at the end of the track, depending on **atend** parameter.
- **double getLength()** – returns the length of the track in meters.
- **ShapePolyLine getPolyline()** – returns the polyline of the track.

Java class Switch

For each circle in the group of rail yard shapes, the **RailYard** object creates a Java object of class **Switch**. **Switch** models a two-way railroad switch that connects three tracks: 0, 1, and 2. Depending on its state, the switch will direct the trains coming from track 0 (*face point movement*) to either track 1 or track 2. The trains coming from track 1 or 2 (*trailing point movement*) will always proceed to track 0 regardless of the state of the switch and will always force the switch to the corresponding state.

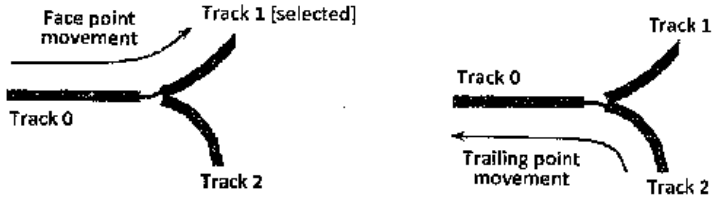


Figure 9.8 A switch

At runtime, the color of a busy switch (the switch with a car over it) is (optionally) different from the color of an idle switch. Colors are set up at the bottom section of **RailYard** parameters. The state of the switch is shown with a thin line drawn over the switch circle. You can toggle the switch state directly by clicking within the circle or via API.

You won't need to use the objects of class **Switch** and their methods too often; the Rail Library object **TrainMoveTo** is capable of setting up the switch states automatically when the train moves along a defined route. In addition, the **RailYard** object has a number of functions like **setSelectedTrack()**, which accept circle names.

However, in some cases the **Switch** API might be useful. To obtain the Switch object that corresponds to a particular circle you should call the function **getSwitch()** of the **RailYard** object, for example:

```
Switch switch123 = railYard.getSwitch( oval123 );
```

Here are some methods of class **Switch** (see *Library Reference Guides: Rail Library* [The AnyLogic Company, 2013] for the full list):

- **Track getSelectedTrack()** – returns the currently selected track (track 1 or 2).
- **setSelectedTrack(Track track)** – select a given track (which should be either 1 or 2). If a car is over the switch, signals error.
- **toggle()** – toggles the selected tracks.
- **Track nextTrack(Track from)** – based on the state of the switch, returns the next track, given the switch is approached from a given track from.
- **boolean isTrailingPoint(Track from)** – tests if movement from a given track from through the switch is a trailing point movement or face point. Returns **true** if trailing point, **false** if facing point.
- **Track getTrack(int index)** – returns the track connected to the switch with a given index.
- **ShapeOval getOval()** – returns the oval shape of the switch.

9.2. Defining the operation logic of the rail model

The operational logic definition in the Rail Library is based on an easy-to-use process flowchart methodology. The entity moving in a rail yard flowchart is a *train* – any sequence of coupled rail cars. There are five flowchart objects in the library that describe all the actions specific to trains:

Object		Description
	TrainSource	Creates trains, performs initial setup and puts them in the yard. Starts any rail yard process flowchart. Supports several types of arrivals scheduling.
	TrainDispose	Removes trains from the model; either those that have exited the yard via an open-ended track, or by "deleting" a train that is still on a track.
	TrainMoveTo	Controls movement of trains. Can calculate routes and set switch states as the train goes along the route. Supports acceleration and deceleration.
	TrainCouple	Couples two trains that "touch" each other into one train. Has two queues and supports coupling of trains on several tracks simultaneously and independently.
	TrainDecouple	Decouples cars from the incoming train and creates a new train from those cars. Supports extreme cases when 0 or all cars are decoupled.

These five objects can be mixed in a flowchart with objects from the Enterprise Library, such as **Delay**, **SelectOutput**, **Hold**, **Seize**, **Release**, **Queue**, etc. The latter are used to define time delays, make decisions, and manage sharing of the rail yard's resources – tracks and switches.

For example, if a part of the yard should be locked to allow a train to pass through, you may associate a resource with it. Then a train that enters that part would need to seize the resource, and the other trains would wait in the queue of the **Seize** object. For the same purpose you can use the **Hold** object and the pair **RestrictedAreaStart/End**.

The **SelectOutput** object can be used in the rail yard process flowcharts to choose between the different process branches, and **Delay** can naturally model the stops durations or duration of operations such as coupling/decoupling or loading/unloading.

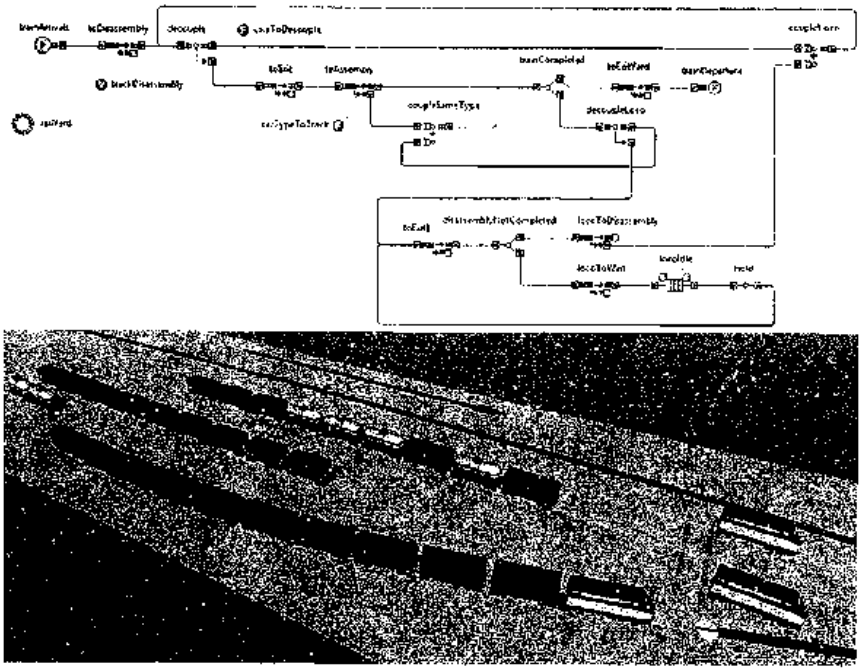


Figure 9.9 A simple classification yard and the flowchart describing its operation

The rail yard operation flowcharts created with AnyLogic Rail Library tend to be very compact. For example, the full logic of a classification yard where arriving trains with different types of cars get disassembled and trains containing cars of the

same type are assembled can be defined by a flowchart with less than 20 objects as shown in Figure 9.9.

Example 9.3: Train stop

We will now create a very simple model with one straight rail track and no switches. Passenger trains will stop in the middle of the track for one minute and then will continue moving in the same direction.

▶ **Create a train and let it move along the track:**

1. Draw a straight polyline with one segment from (0,50) to (1100, 50). Call it **track**.
2. Right-click the polyline and choose **Grouping | Group** from the context menu. A group containing just that polyline is created. Call it **groupRailYard**.
3. Open the **Rail Library** palette and drag the **RailYard** object to, say, (50, 100).
4. Set the parameter **Rail yard shapes** of the **railYard** to **groupRailYard**.
5. From the same palette drag the three objects: **TrainSource**, **TrainMoveTo**, and **TrainDispose**. Place them in a sequence and connect as shown in Figure 9.10 (to connect two ports double-click one of them, drag the connector to the other port and click there).
6. Select **trainSource**. Set the following parameters of that object:
   Rail yard object: railYard
   Track (polyline): track
   Offset of 1st car, meters: 200
   Cruise speed, meters/sec: 10
7. Click on the model item (the topmost one) in the **Project tree**. Check that the **Time units** parameter is set to **minutes** in the **General** page of its properties.
8. Run the model. Until the model time is 10, nothing happens. At 10 (and then at 20, 30, etc.) a train appears at the beginning of the track, moves to the right and exits the track.

Look at the parameters of **trainSource**.

We have set up the **trainSource** to create a train in our rail yard and place it on the **track** with the front side of the first car located 200 meters from the beginning of the track. The train will have forward orientation relative to the track (parameter **Orientation on track**), so the rest of train will be between the beginning of the track and point 200. At the time of creation, the train must be fully on the track, so you need to make sure there is enough space. By default, a train created by **TrainSource** has 11 cars, and the default length of a rail car is 14 meters. As $11*14 = 154 < 200$, there is enough space.

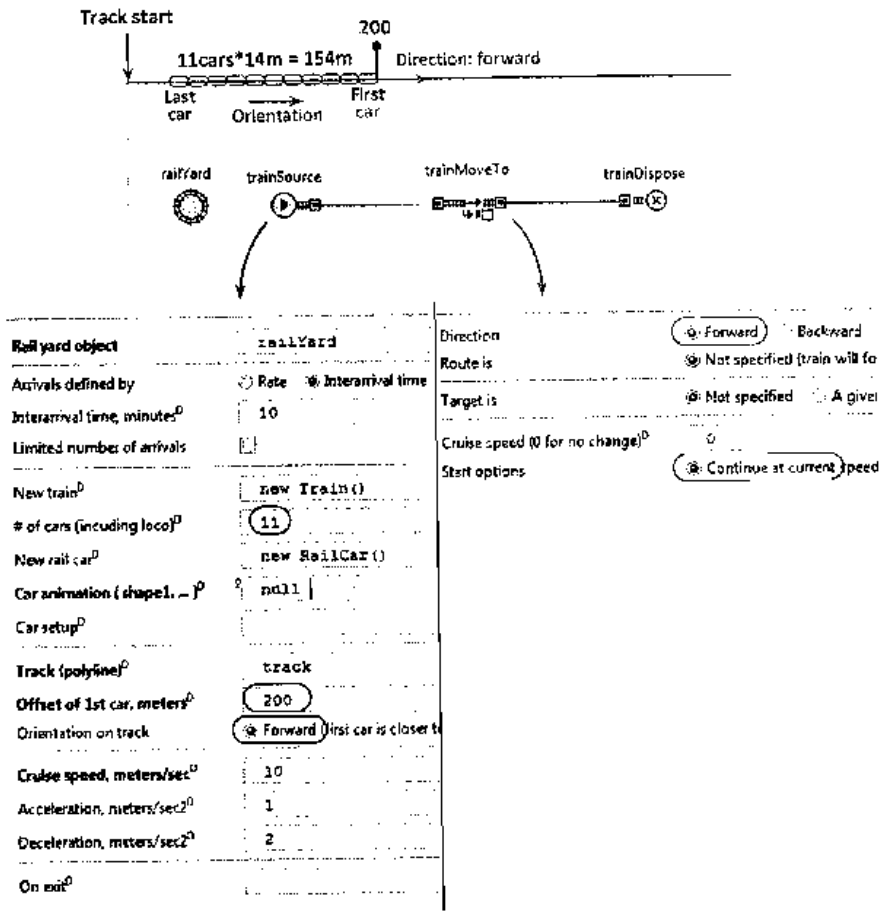


Figure 9.10 The simplest train process flowchart: train appears and moves along the track

The default arrival schedule of **TrainSource** is one train every 10 minutes, so at times 10, 20, 30 and so on. Remember, the time units are meters. Our **trainSource** will create a new train at the same place, and we need to make sure the previous train has moved and gotten out of the way.

The *cruise speed* that was specified in **trainSource** (10 meters per second, which equals 36 km/h) will be the default speed for train movement.

The train has a *current velocity* (the actual velocity at which it moves at the current moment of time). At the time of creation by **TrainSource** the current velocity of a train

is set to its cruise speed as if the train has just been moving with the cruise speed. This is done to enable immediate continuation of movement at the same velocity. You can change the velocity at any time by calling the function **setVelocity()** of the train.

Now look at the parameters of **trainMoveTo**. You will see that the **Start options** is set to **Continue at current speed**, and the **Direction** is **Forward**, which means the train will appear in the model moving to the right at 36 km/h. No target and no routing options are specified in **trainMoveTo**, so the train will naturally exit the **track** at its right end and the train entity will exit the **trainMoveTo** object and will be consumed by **TrainDispose**.

Now we will add a short train stop.

▶ **Add the train stop**

9. Open the Presentation palette, double-click the line object and draw a short vertical line crossing the track at X=500. Call the line **redLine**.
10. Select **trainMoveTo** object and set the following parameters:
    Route is: Calculated automatically from current to target track
    Target is: Intersection of a given track and a line
    Track: track
    Line: redLine
11. Run the model. You will see that now trains disappear once they reach the red line.
12. Modify the flowchart. Disconnect the **trainDispose** object from **trainMoveTo** and move it to the right to give space for two new objects.
13. Rename the **trainMoveTo** object to **toStop**.
14. Open the **Enterprise Library** palette and drag the **Delay** object in the flowchart after **toStop**. Call the object **trainStop**. Set its parameter **Delay time to 1 * minute()**
15. Open the **Rail Library** palette again and drag another **TrainMoveTo** object between **trainStop** and **trainDispose**. Call it **toExit**. Connect the ports of all objects.
16. Run the model again. Now the train stops at the red line, waits there for one minute, and then continues to the right end of the track.

In the first **TrainMoveTo** object, which is now called **toStop**, we specified the target position of the movement, which is graphically defined by the red line. (Note that the line is *not* a part of the rail yard shapes group.) Having reached the red line the train entity exits the **toStop** object and enters the object **trainStop** of type **Delay**. While the train entity waits in the **trainStop** object, the train does not move. Then, after a

one minute delay, the train entity exits **trainStop** and enters the second **TrainMoveTo** (**toExit**) which moves it further.

In this flowchart we mixed the objects from the Rail Library with the objects from the Enterprise Library. This is possible because the **Train** object that is generated by **TrainSource** is a subclass of **Entity** -- a generic entity that can be handled by process flowchart objects from the Enterprise Library.

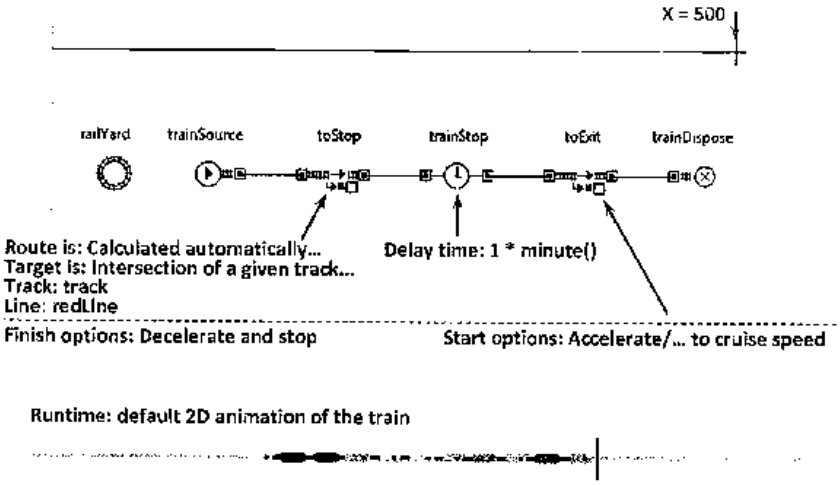


Figure 9.11 The complete process flowchart for the Train stop model. Default 2D animation.

If, however, you look into the parameters of **toExit**, you will see that **Start options** is set to **Continue at current speed**. So you may ask: why did the train start moving when its speed had been 0 at the train stop? The answer is: we did not ask the train to decelerate and stop during the first movement. Therefore, the train exited **toStop** object with its original velocity (10 meters/sec), which "remained set" during the stop, although the train was not physically moving. This is obviously an unrealistic behavior: the train cannot stop immediately and cannot immediately gain speed. This is not an accurate representation of the physics of trains, though it may be useful for certain models with a higher level of abstraction. Now, however, let's add deceleration before the stop and acceleration afterwards.

▶ **Add acceleration / deceleration**

17. Select **toStop** and set **Finish options** to **Decelerate and stop**.
18. Select **toExit** and set its **Start options** to **Accelerate/decelerate to cruise speed**.

19. Run the model. Play with different cruise speed, acceleration and deceleration values (they are set up in **TrainSource**). Try to slow down the simulation.

As a final part of this example, we will add the 3D animation.

▶ **Add 3D animation**

20. Click the **groupRailYard** and select **Show in 3D scene** on the **General** page of its properties.
21. Right-click the group and choose **Select group contents** from the context menu. All rail yard shapes get selected.
22. In the **Advanced** property page of the multiple selection set **Z-Height** to 1.
23. Open the **3D** palette and drag the **3D Window** to the canvas below the flowchart. Change its size to, say, 1000x250 pixels.
24. Run the model. Move the camera to view the track and the trains better.

The default 3D animation of trains is very schematic. However, you can always use a custom 3D object as animations of your rail cars. Some ready-to-use objects are contained in AnyLogic **3D Objects** palette.

▶ **Add custom 3D objects for the rail cars**

25. Open the **3D Objects** palette. Drag the **Passenger Car** and the **Locomotive** to anywhere in the canvas.
26. Select **trainSource**.
    Set the parameter **Car animation {shape1, ... }** to **{locomotive, passengerCar}**.

Although there are 11 cars in our train, we can provide only two shapes in the list: the last shape will be used for all remaining cars. This is fine in our case: all cars except for the first one are passenger cars.

27. Run the model. View the 3D animation of the train. Notice that the cars heavily intersect.

It happens because the 3D objects for cars have different lengths (as real cars do), but **TrainSource** uses just one default length of the **RailCar**, which is 14 meters. We need to specify the custom lengths of our cars. This can be done in the parameter **Car lengths, meters** of **TrainSource** or in the **Car setup** code.

28. Select **trainSource**. Set the parameter **Car lengths, meters** to **{14, 27}**.

In the AnyLogic **3D Objects** palette the locomotive and all types of freight cars have a length of 14 meters, and the passenger car is 27 meters long.

29. As our train is now longer, we need to adjust its initial location so all cars fit on the track. Set the parameter **Offset of 1st car, meters** to: **14 + 27*10 + 10**. (The last 10 is just for safety: numeric errors can always occur.)
30. Run the model.

▶ **Draw the platform at the train stop**

31. Open the 3D palette and drag **Rectangle** to the canvas.
32. Edit the rectangle size and position this way (you may need to change zoom):
    X: 265, **Y:**52, **Width:** 285, **Height:** 10. The rectangle should appear just below the track to the left of the red line.
33. Set the **Line color** of the rectangle to **No color** and **Fill color** to **concrete** texture.
34. On the Advanced page of the rectangle properties set:
    Z: 2
    Z-Height: 1.
35. Run the model again.

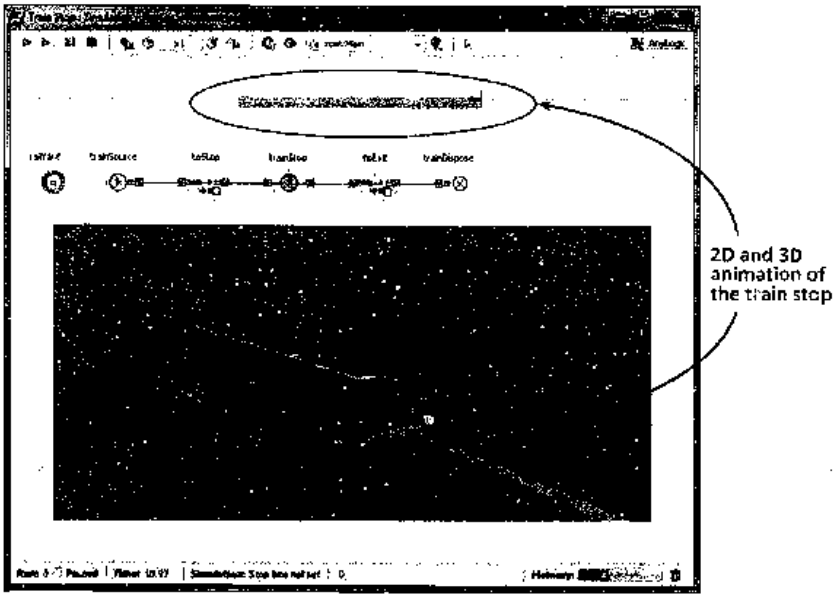


Figure 9.12 A screenshot of the Train Stop model with an adjusted camera position

Example 9.4: Ensuring safe movement of trains

The AnyLogic Rail Library enables you to define rail topology and control the movement of trains but ensuring the safe train movement is the task of the modeler.

The library will only detect train collisions at switches (a train arrives at a switch while another train is over the switch) or when a train collides with another train moving or standing on the same track.

In this example, we will show how you can use AnyLogic Enterprise Library resources to model a simple safety control system. The implementation of the safety control in a real rail yard will contain various communication protocols between the train and the dispatchers, traffic lights, and other elements that are outside the Rail Library's scope. Here we are only suggesting one of the ways to map train management policy to AnyLogic modeling language.

We will use the rail layout from Example 9.1: "A very simple rail yard" considered earlier in this chapter. Assume trains are arriving from the left every 5 minutes. Every third train goes to the lower track, stays there from 3 to 10 minutes and then continues in the same direction. All other trains just move along the main track from left to right. As the trains merge at the switchCBD, collisions are possible and we need to manage the traffic to avoid the collisions.

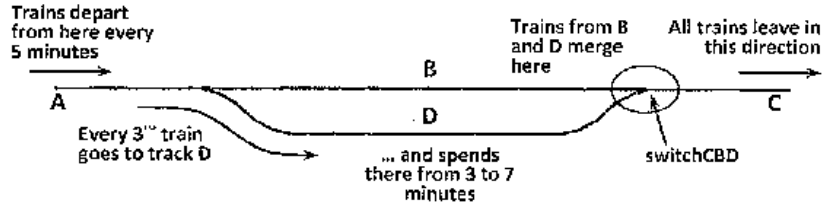

Trains depart from here every 5 minutes
Trains from B and D merge here
All trains leave in this direction
B
A
C
D
Every 3rd train goes to track D
... and spends there from 3 to 7 minutes
switchCBD

Figure 9.13 The problem definition for the example Ensuring safe movement of trains

First we will define the traffic without the safety control.

▶ **Create the rail configuration and set up the traffic without safety control:**

1. Repeat the steps 1-19 of Example 9.1: "A very simple rail yard" (or just copy the group of the rail yard shapes and the **railYard** object from there).
2. Open the Presentation palette and draw two short vertical lines: **redLine** crossing the **trackB** at X=800, and **blueLine** crossing **trackD** at X=750 (see Figure 9.14).
3. Open the **Rail Library** palette and drag the **TrainSource** object to the canvas at, say, (150,250). Set the following parameters of the trainSource:
   Rail yard object: **railYard**
   Interarrival time, minutes: **5**
   # of cars, including loco: **8**
   Track (polyline): **trackA**
   Offset of 1st car, meters: **120**
   Cruise speed, meters/second: **10**

4. Open the **Enterprise Library** palette and drag the **SelectOutput** object to the right of **trainSource**. Call it **notEachTrhird** and change its parameters:
   Select True output: If condition is true
   Condition: self.in.count() % 3 != 0

We need to treat each 3rd incoming train differently, and this **SelectOutput** object is here to distinguish between each 3rd train and all other trains. The easiest way to do it is to find out how many trains have entered the **SelectOutput**, divide it by three, and look at the remainder. For every 3rd train the remainder will be 0.

In the dynamic parameters of embedded active objects you can always access the variable **self** – this is the embedded object itself. So, the expression **self.in.count()** written in a dynamic parameter of **notEachTrhird** returns the number of entities that have entered **notEachTrhird** via its **in** port.

5. Continue the flowchart by adding two **TrainMoveTo** objects after the two outputs of **notEachTrhird**.
6. Rename the **TrainMoveTo** at the T (true) output port of **notEachTrhird** to **moveToRedLine** and set the following parameters:
   Route is: Calculated automatically...
   Target is: Intersection of a given track and a line
   Track: trackB
   Line: redLine
7. Similarly, rename the **TrainMoveTo** at the F (false) port to **moveToBlueLine** and set the following parameters:
   Route is: Calculated automatically...
   Target is: Intersection of a given track and a line
   Track: trackD
   Line: blueLine
8. Add a **Delay** object from the Enterprise Library after **moveToBlueLine** and set its **Delay time** to **uniform(3, 10)**.
9. Add the third **TrainMoveTo** and connect the outputs of **moveToRedLine** and **delay** to its input. Call it **moveToExit**.
10. Finish the flowchart with **TrainDispose**.
11. Run the model. Watch the trains going through the yard. The first several trains may do fine, no collisions occur.
12. Run the model in virtual time mode (as fast as possible). The model will stop almost immediately with an error "Switch 'switchCBD' cannot change its state while a car is over it".

As long as the time the trains spend at **trackD** is nondeterministic and can be greater than the time interval between trains (5 minutes), collisions at **switchCBD** will inevitably occur. The error message you are getting (see Figure 9.14) is typical for that type of collision: while the train exiting **trackD** is still moving over the **switchCBD**, another train from **trackB** approaches the switch and tries to change its state (in this case as a result of trailing point movement).

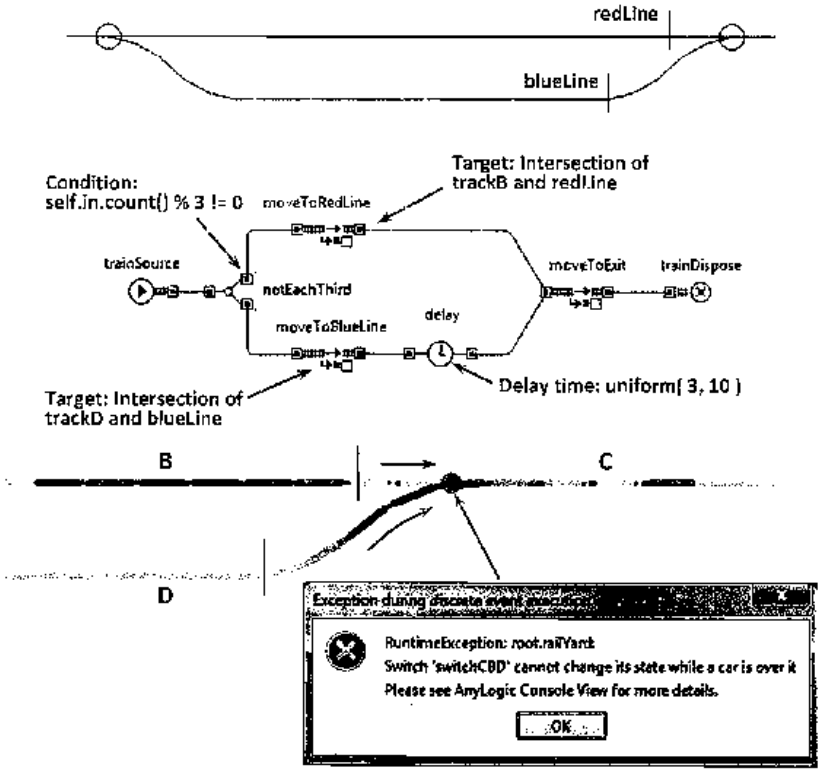


Figure 9.14 The unsafe train movement flowchart and the collision detected by the library

We will now add a safety management mechanism to our model. The idea is to associate an Enterprise Library resource with the part of the rail yard shared by the conflicting train flows and have trains seize the resource before they enter that part and release it when they exit.

▶ **Add safety control**

13. Modify the flowchart by embracing the **moveToExit** object (the one where conflict actually occur) with the pair **Seize** / **Release**, both connected to a **ResourcePool** as shown in Figure 9.15. Those three objects can be found in the **Enterprise Library** palette. Leave all parameters with at default values.
14. Run the model again. Try virtual time mode. No collisions occur.

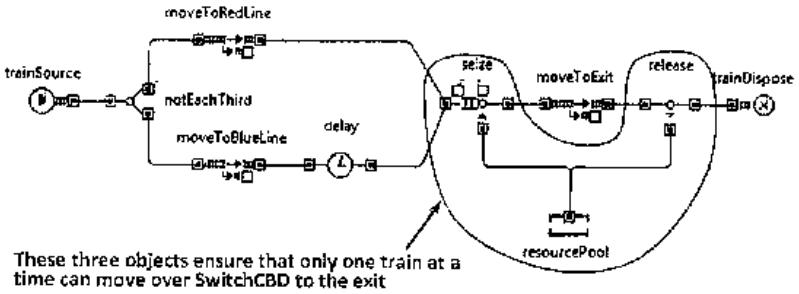


Figure 9.15 The problem definition for the example Ensuring safe movement of trains

Associating **ResourcePool** with a shared section of a rail yard is not the only way of avoiding conflicts. Alternatively you could use the object **Hold** or a pair **RestrictedAreaStart**/**End**.

The model of the safety control system is now in place. You may have noticed, however, that the movement of trains is not very realistic: they start and stop without acceleration and deceleration, the speed changes from 0 to 10 m/s (cruise speed) immediately. You already know that you can request **TrainMoveTo** to apply acceleration and deceleration. However, in this particular example, trains that move along the main track (A-B-C) should only decelerate before the **redLine** if the exit is seized by another train; otherwise they should just continue going at cruise speed. How can we implement such optional deceleration?

One approach is to determine whether the exit is busy (seized by another train) at some point before the **redLine**. If it is busy, the train will decelerate and stop at the **redLine**, where it will wait for the resource to become available. If the exit is free, the train will immediately seize it and proceed to the exit without deceleration.

▶ **Add acceleration and deceleration of trains:**

15. Draw one more line crossing **trackB** at X=700. Change its color to yellow and call it **yellowLine**.

16. Insert two objects between the **true** port of **notEachThird** and the input port of **moveToRedLine: TrainMoveTo** and **SelectOutput** as shown in Figure 9.16. Call them **moveToYellowLine** and **exitIsBusy**.
17. Set the following parameters of **moveToYellowLine**:
    Route is: Calculated automatically...
    Target is: Intersection of a given track and a line
    Track: trackB
    Line: yellowLine
18. Change the parameters of **exitIsBusy**:
    Select True output: if condition is true
    Condition: resourcePool.idle() == 0
19. Change the start and finish options of the three **TrainMoveTo** objects created earlier:
    moveToRedLine: Finish options: Decelerate and stop
    moveToBlueLine: Finish options: Decelerate and stop
    moveToExit: Start options: Accelerate/decelerate to cruise speed
20. Run the model. You may need to slow down the simulation to see how the trains accelerate.

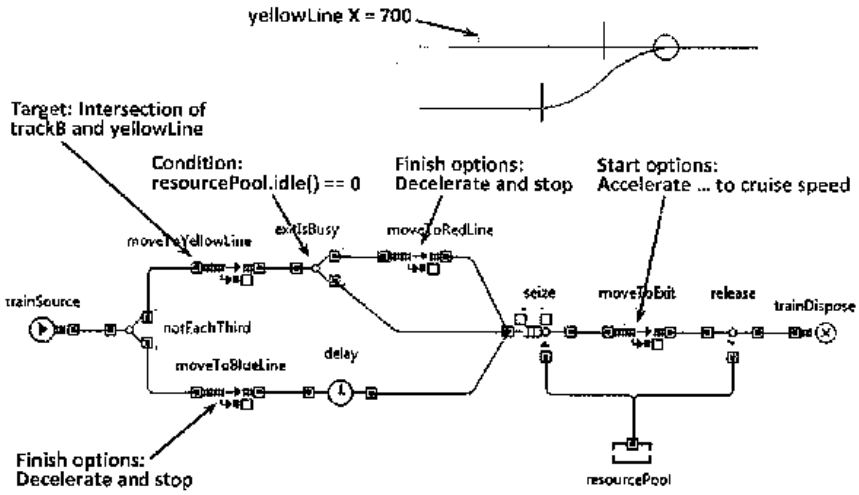


Figure 9.16 The flowchart modified to model acceleration and deceleration

To check the state of the exit area of the rail yard in advance, we added an intermediate point on **trackB** marked with the **yellowLine**, and then we divided the movement into two stages: move to the **yellowLine** (without deceleration), and if the exit is idle, seize it immediately and proceed to the exit without deceleration.

Otherwise (if the exit is busy), decelerate, stop at the **redLine** and wait there until the exit gets free. Note that:

- Continuous movement can be modeled by multiple **TrainMoveTo** objects put in a sequence (like the movement along the main track A-B-C is modeled by **moveToYellowLine** and **moveToExit**).
- **TrainMoveTo** with the start option **Accelerate/decelerate to cruise speed** will either take a train already going at its cruise speed (in which case acceleration will not be done), or will accelerate a standing train – see **moveToExit**.
- The same **TrainMoveTo** object can take trains at different locations and bring them to the same or different destinations. For example, a train entering **moveToExit** can be either at a red, blue, or yellow line, but will be routed to the exit.

Example 9.5: Simple classification yard

In this example, we will begin by creating a very simple classification yard. Arriving trains will contain cars of two types: tanks and box cars. The tanks will be moved to the departure track and left there, and the box cars will continue to the exit. Once the departure track accumulates more than six tanks, they will be driven away by another locomotive. Once again, we will use the rail layout created in Example 9.1: "A very simple rail yard".

Next, we will create a custom rail car class with information about the car type and with built-in statistics, and we will show how to access the properties of individual rail cars. We will also show how to *couple* and *decouple* rail cars.

▶ **Create a custom class for rail cars:**

1. Repeat the steps 1-19 of Example 9.1: "A very simple rail yard" (or just copy the group of the rail yard shapes and the **railYard** object from there).
2. In the **Projects** tree right-click the model (topmost) item and choose **New | Java class.**
3. In the new Java class wizard enter the name of the new class: **FreightCar** and choose the base class **com.xj.anylogic.libraries.railyard.RailCar** (this is available in the drop-down menu). Press **Next.**
4. Add two fields: **type** of type **int** and **timeEntered** of type **double** (see Figure 9.17). Press **Finish.** The Java editor opens for the new class. We do not need to modify the class implementation, so you can close the editor.
5. Return to the editor of **Main** where you have the rails and the **railYard** object.
6. Define three integer constants for three car types. To do it open the **General** palette and drag the **Variable** item to the canvas. Call it **LOCO** and set the following properties:

Static: yes
Constant: yes
Type: int
Initial value: 0

7. Ctrl+drag the **LOCO** constant to create a copy. Call the copy **BOX** and set its initial value to **1**.
8. In the same way create the third constant **TANK** with value **2**.

It is Java code convention to capitalize all letters in constants' names. This way you can easily distinguish them from variables.

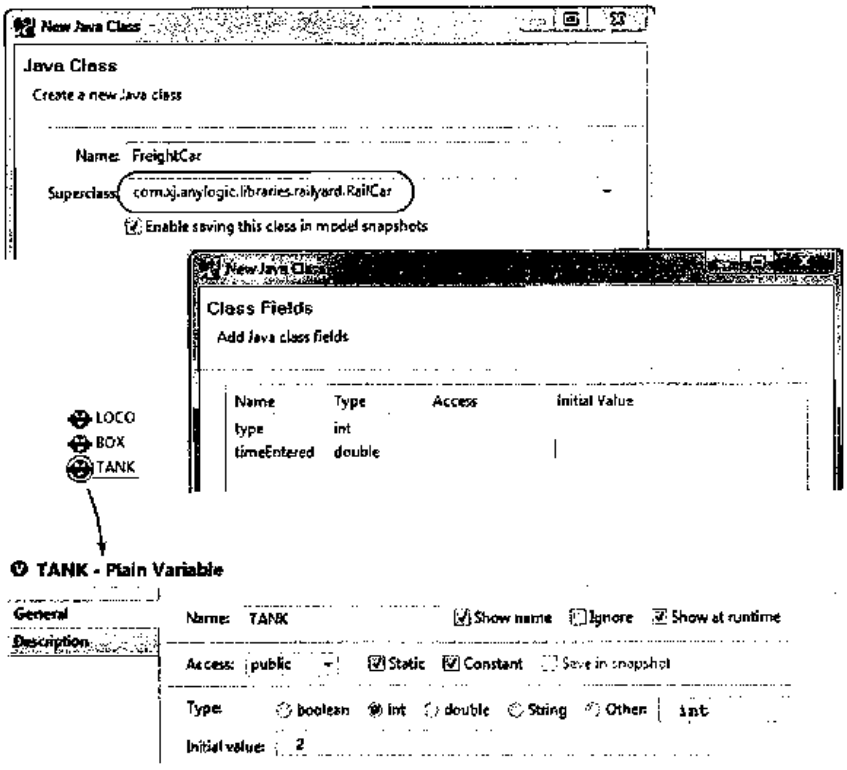


Figure 9.17 A custom class for rail cars and three constants for three car types

▶ **Create a simple train flowchart and set up train creation:**

9. Open the **3D Objects** palette and drag three objects from there: **Locomotive**, **Box Car**, and **TankCar**. Set the scale of all three objects to 200% (remember that the **Scale** parameter of the **railYard** is set to 2 pixels per meter).

10. Create a flowchart of three Rail Library objects: **TrainSource**, **TrainMoveTo**, and **TrainDispose**.
11. Set the following parameters of **trainSource**:
    Rail yard object: railYard
    # of cars (including loco): 6
    New rail car:
        new FreightCar(carIndex == 0 ? LOCO : (carIndex < 3 ? TANK : BOX), time())
    Car animation: { locomotive, tankCar, tankCar, boxCar, boxCar, boxCar }
    Track polyline: trackA
    Offset of 1st car: 100
    Cruise speed, meters/sec: 5
12. Run the model. At time 10 (and then at 20, 30, and so on) a new train appears on the main track and proceeds to the right.

The rail cars in our train are a custom class, **FreightCar**,, which is a subclass of **RailCar**. This means that the cars have all of the properties of the base class as well as the properties we defined for **FreightCar**, namely the fields **type** and **timeEntered**. In the field **New rail car** of **TrainSource**, we call the default constructor of **FreightCar** with two parameters: type and time of creation. The type depends on the position of the car in the train, which can be accessed as **carIndex**. The first car (position 0) is the locomotive (constant **LOCO**), followed by two tank cars and three box cars. The second parameter is set to the current model time, the function **time()**.

▶ **Model decoupling of tank cars and moving them to the departure track**

13. Open the **Presentation** palette and draw two short vertical lines: **redLine** crossing the **trackB** at X=800, and **blueLine** crossing **trackD** at X=450 (see Figure 9.18).
14. Modify the flowchart as shown in Figure 9.18.
15. Run the model (as the process flowchart is incomplete, it makes sense to watch the first train only: the following trains will just crash into the box cars standing on track B).

The object after **TrainSource** is **toRedLine** (of type **TrainMoveTo**); it brings the incoming train to the **redLine**. The next object, **decoupleTanks** (type **TrainDecouple**), decouples 3 cars: the loco and two tanks. Next, **tanksToC** takes the decoupled part of the train to the **trackC**. Finally, the last object **tanksToD** pushes the loco and the tanks to **trackD** where they stop at the blue line.

The object **TrainMoveTo** can only calculate the straight routes, i.e. the routes not requiring reverse movement of the train. Therefore, to bring the tanks from B to D we need to use two **TrainMoveTo** objects: one that moves the train forward to C (note

that the offset on C is chosen to fit the full train plus 10 meters), and another that moves the train backwards to track D.

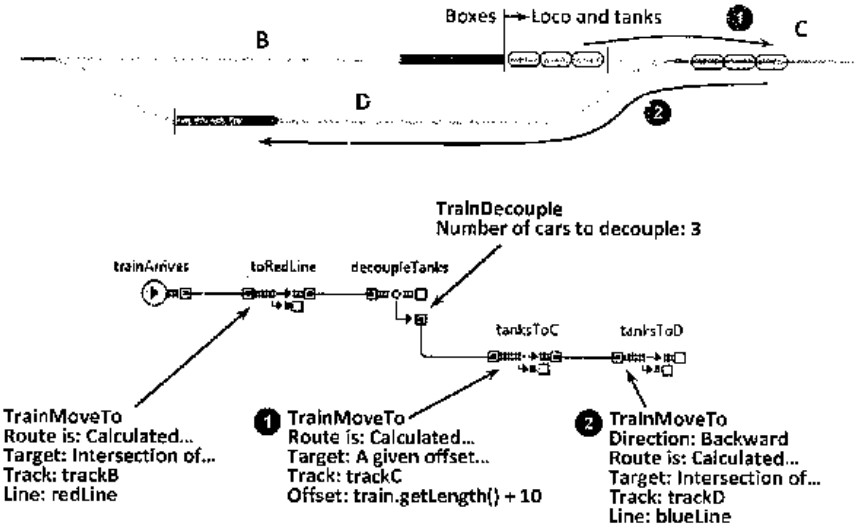


Figure 9.18 The first step of the classification process: tank cars are decoupled and moved to D

In the object **decoupleTanks**, we assumed that any train has only two tank cars after the locomotive. We will now generalize that object so that is will decouple any number of tank cars. To accomplish this, we need to access the contents of the incoming train and look into the properties of the individual cars.

▶ **Generalize TrainDecouple to handle any number of tank cars**

16. Create a new function with the name **nCarsToDecouple** (**Function** object is in the **General** palette). Set the **Return type** of the function to **int** and add one parameter **train** of type **Train** (remember that AnyLogic is case sensitive).
17. Write the following code in the body of **nCarsToDecouple**:

```
int n = 1; //the first cars is locomotive
while( ((FreightCar)train.get(n)).type == TANK )
        n++;
return n;
```

18. Set the parameter **Number of cars to decouple** of **decoupleTanks** to: **nCarsToDecouple(train)**
19. Run the model. The behavior is the same. Try to modify the parameters of **trainSource** so that the incoming trains have different or variable number of tank cars following the locomotive.

In the code of **nCarsToDecouple**, we access the object of class **Train**. The method **get(n)** returns the rail car with a given index. However, the class of the returned rail car is a generic class **RailCar**, which we extended in order to include information about the car type. We therefore need to "cast" it to **FreightCar** to access the type field. In the field **Number of cars to decouple** we call the function **nCarsToDecouple**, giving it the current train (**train**) as an argument.

We will now continue the classification process. Having moved the tanks to track D, the locomotive will decouple from them, return to track B, couple with the box cars waiting there, and leave the yard.

▶ **Model return of locomotive to box cars and their departure from the yard:**

20. Add more objects to the flowchart and set their parameters as shown in Figure 9.19. Leave the parameters of the last three objects **coupleBoxes**, **boxesToExit** and **boxesExit** at their default values. Note that the output port of **locoToBoxes** is different: it is **outHit** port.
21. Run the model. The locomotive now returns to box cars and drives them away. Again, the flowchart is still incomplete and already the second incoming train will not be able to complete the process correctly.

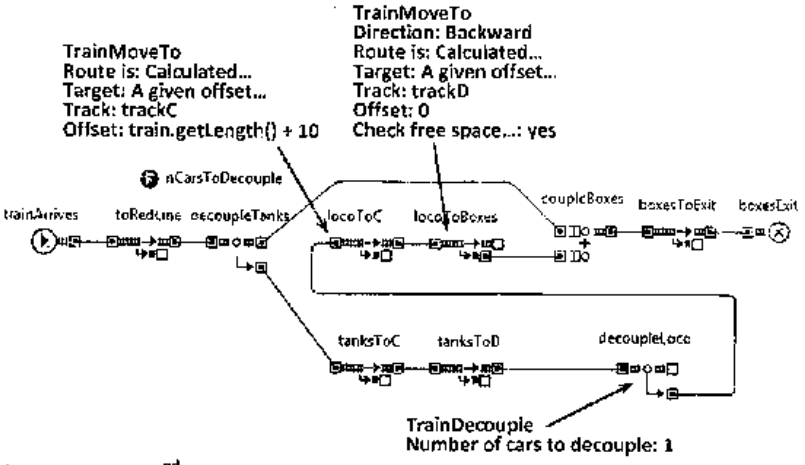


Figure 9.19 The 2nd step of classification: loco returns to box cars, picks them up, and drives away

Decoupling the locomotive is quite straightforward and can be accomplished in the following steps. First, decouple the first car of the train. Then, return to the box cars on track B, utilizing two moves: **locoToC** and **locoToBoxes**. The parameters of **locoToBoxes** are described as follows. The beginning of the track C (offset 0) is set as the target point. We know that there are box cars on track C, so the loco will

inevitably hit them on its way to the target point. To let the locomotive move as close as possible to the nearest box car and then stop, we selected the parameter **Check free space on target track** of **TrainMoveTo**.

The option **Check free space on target track** allows you to finish movement at the first rail car met on the target track. The train would "touch" the car and stop. This option is mostly used when you are going to couple with that car. Please keep in mind that the free space on the target track is checked *at the time the train starts movement*, so if the situation on the track changes while the train is moving, it may stop at an incorrect position. If the train stops having touched another train, the train entity exits via **outHit** port of **TrainMoveTo**, otherwise the train proceeds to the target position, and exits via **out** port.

Next, look at the object **coupleBoxes** of type **TrainCouple**. It has two input ports for the two parts of the train that will be coupled into one train. When a train arrives to either of the two inputs, **TrainCouple** checks to see whether it "touches" any train waiting at another input. If yes, the trains are coupled into one (more precisely, the train at **in2** is added to the train at **in1**), and the train entity exits **TrainCouple**. Otherwise, the train waits in the queue associated with the input port.

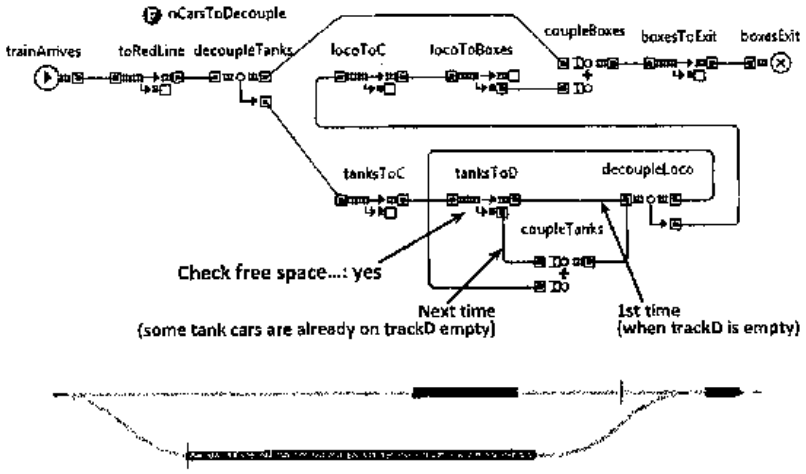


Figure 9.20 The tank cars are coupled on the departure track

Now we will show how to extend the classification process to couple the tank cars that are moved to **trackD** with other tank cars that are already there.

▶ **Model coupling of tank cars with each other:**

22. Add one more object to the flowchart: **coupleTanks** of type **TrainCouple** and connect it as shown in Figure 9.20.
23. Set the parameter **Check free space on target track** of tanksToD to yes.
24. Run the model. The tank cars now accumulate on the departure track and are coupled with each other.

The departure track has limited capacity and the cars that have accumulated on it need to be moved out of the way periodically. Another locomotive will be brought in as soon as eight tank cars have accumulated on the departure track. The locomotive will come from the left-hand side (track A), couple with the tanks, and drive them away back to the left.

▶ **Model departure of the tank cars**

25. Modify the flowchart as shown in Figure 9.21.
26. Run the model. The tanks are now periodically driven away by a new locomotive and the model can run for an infinite amount of time.

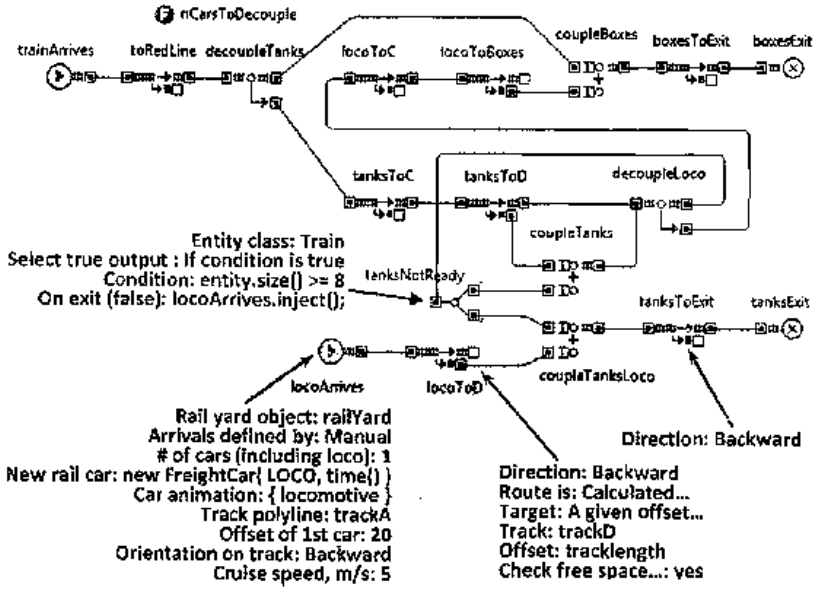


Figure 9.21 The final version of the classification flowchart

Note that in the flowchart object tanksNotReady (of type SelectOutput) we need to determine the size of the train, i.e., we need to access the property specific to entities of class Train. As SelectOutput is an Enterprise Library object and not a Rail

Library object, we need to tell it that entities going through it are of class **Train** – this is done by specifying Train in the **Entity class** field. Only after this is completed can we write **entity.size()**.

The arrival of locomotives that drive away the tanks is triggered from the **SelectOutput** object **tanksNotReady**, as seen in the action code of its **On exit (false)** field. Correspondingly, the arrival type of **TrainSource** is set to **Manual**.

There are a couple of additional things to notice in this flowchart. First, although the tank cars are all physically accumulated on track D, in the flowchart they are waiting either in the queue of **coupleTanks** object, or (if there are eight of them) in the queue of **coupleTanksLoco** object. Second, notice the orientation of the locomotive and the departure train. The initial orientation of the locomotive is backward (parameter **Orientation on track** of **locoArrives**), such that it couples with the tanks at its rear side. After coupling, however, the loco is added to the tanks and not vice versa because it enters the **TrainCouple** object from its lower port. Therefore, the orientation of the departure train is same as orientation of the tank cars, and departure to the left is departure in backward direction – see the parameter **Direction** of the **tanksToExit**.

Now that the classification process is complete, we can collect some statistics. Let us obtain the length of stay of the tank cars in the yard. You may remember that we have recorded the time each car entered the yard by providing the current model time in the constructor of the **FreightCar** class – see the parameter **New rail car** of the **trainArrives** object. Now we will calculate the length of stay at the time the cars are exiting the yard. We will use the field **On exit yard** of the **railYard** object.

▶ **Add collection of length of stay statistics**

27. Open the **Analysis** palette and drag the **Histogram data** object to the canvas. Call it **lengthOfStay** and set the **Number of intervals** to 20.
28. Select the **railYard** object and set the following parameters:
    Rail car class: **FreightCar**
    On exit yard:

```
if( car.type == TANK )
        lengthOfStay.add( time() - car.timeEntered );
```

29. Drag the **Histogram** chart from the **Analysis** palette to the canvas.
30. Click **Add histogram data** button on the **General** page of its properties and enter **lengthOfStay** in the **Histogram** field.
31. Run the model. Switch to virtual time model so that statistics is collected faster.

The length of stay appears to be deterministic and takes one of the four values, as shown in Figure 9.22. This is expected: the model itself is a deterministic model, and the variation of length of stay is due to the fact that tank cars are moved to the departure track in four portions of two cars each, so the first two rail cars will wait for a longer time than the last. You may add stochastic elements to the model, such as:

- Variation of arrival times of the original trains or the locomotive
- Stochastic time delays associated with coupling and decoupling
- Cruise speed variations

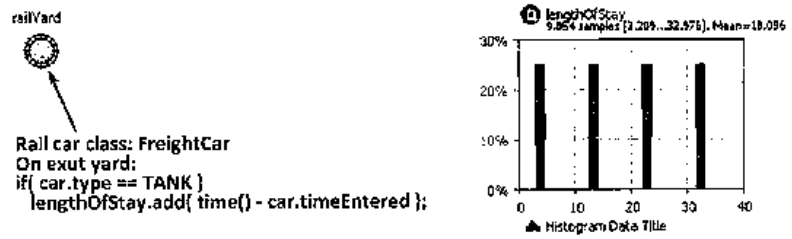


Figure 9.22 Collecting length of stay statistics for the tank cars

▶ **Add acceleration, deceleration and 3D animation of the model**

32. In all **TrainMoveTo** objects with the exception of **toRedLine** and **locoToD** set **Start options** to **Accelerate/decelerate to cruise speed.**
33. In all **TrainMoveTo** objects without exception set **Finish options** to **Decelerate and stop.**
34. Open the **3D** palette and drag the **3D Window** to the canvas below the flowchart. Change its size to, say, 750x350 pixels.
35. Run the model. Move the camera to find a better viewpoint.

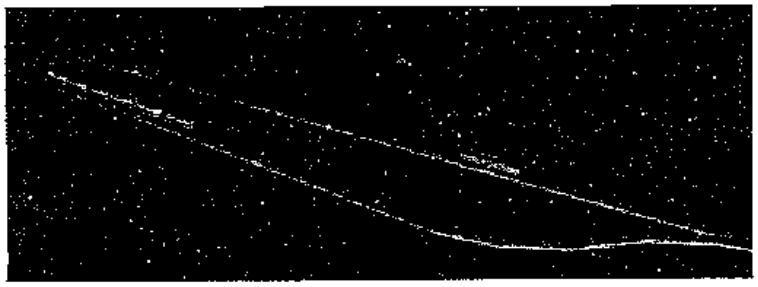


Figure 9.23 3D animation of the classification yard

Example 9.6: Airport shuttle train (featuring AnyLogic Pedestrian Library)

We will now make the Rail Library and the Pedestrian Library work together by modeling a shuttle train that transfers passengers between two airport terminals. The passengers will be modeled using the AnyLogic Pedestrian Library when they are on the platform, and when they are on the train, they will be contained in a **Train** entity. The rail system will be very simple: just one passenger car moving back and forth along one straight track.

▶ **Model the rail part of the system:**

1. Draw a straight one-segment polyline from (50,50) to (950,50). Call it **track.** Select its property **Show in 3D scene.**
2. Create a group containing just this polyline and call the group **groupRails.**
3. Drag the **RailYard** object from the **Rail Library** palette and type **groupRails** in its **Rail yard** field. Also set the **Scale, pixels per meter** to 5.
4. Open the **3D objects** palette and drag the **Passenger Car** object anywhere on the canvas. Set the scale of all three objects to 500% (corresponds to **Scale** set to 5 pixels per meter).
5. Create the train process flowchart as shown in Figure 9.24. Note that the delay time "1" means one minute as the model time units are minutes (default setting, see the **General** page of the model properties).
6. Run the model.

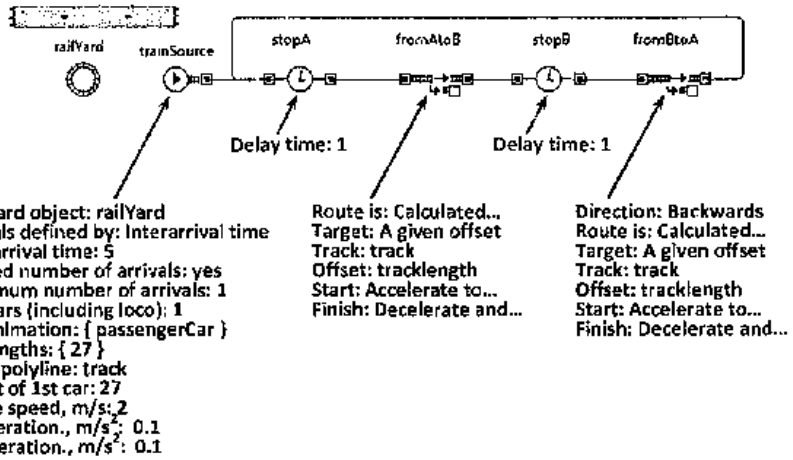


Figure 9.24 The rail part of the Airport shuttle train flowchart

Note that the **TrainSource** object in this model is used to create only one train that remains in the system. We have prevented any generation of other trains by setting the **Maximum number of arrivals** to 1.

▶ **Mark up the pedestrian ground and create configuration objects:**

7. Create the markup as shown in Figure 9.25. Use **Line** and **Rectangle** objects from the **Presentation** palette.
8. Drag **PedConfiguration** and **PedArea** objects from the **Pedestrian Library** palette. Call **PedArea waitAreaA** and set the properties of these two objects as shown in Figure 9.25.

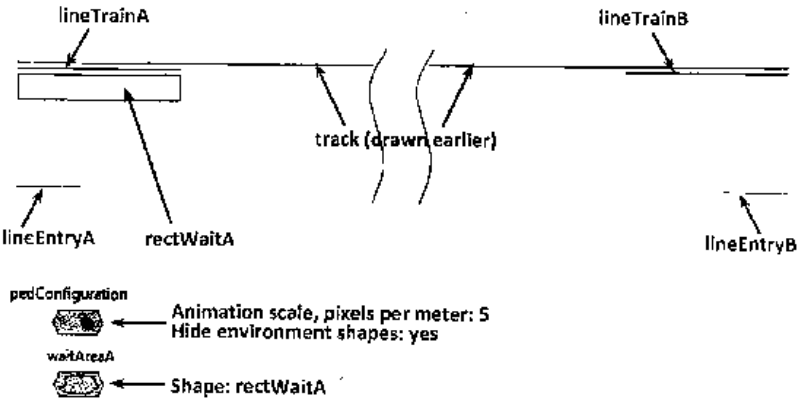


Figure 9.25 Markup of the pedestrian ground

▶ **Add the pedestrian part of the flowchart:**

9. Insert **Pickup** and **Dropoff** objects into the rail flowchart as shown in Figure 9.26. Set their **Pickup/Dropoff** parameters to **All available entities.**
10. Add the pedestrian flowchart and set the parameters of the objects as show in Figure 9.26.
11. Add the entry/exit action of the **stopA** object as shown.
12. Run the model.

Once the train arrives at the stop at terminal A, it lets the waiting passengers in by calling the method **freeAll()** of the **waitA** object (of type **PedWait**). Upon closing the doors, the passengers who could not board the train are returned to the wait area by calling the method **cancelAll()** of the object **boardA**. Then they exit the object via the port **cl** and get back into **waitA**.

The object **exitPedModelA** of type **PedExit** temporarily removes the passengers who boarded the train from the pedestrian ground, i.e. from under control of the Pedestrian Library. The passengers are then picked up by the train entity and travel with the train to the terminal B, where they unboard (object **dropoffB**), are returned to the control of the Pedestrian Library, and appear again on the pedestrian ground at **lineTrainB**.

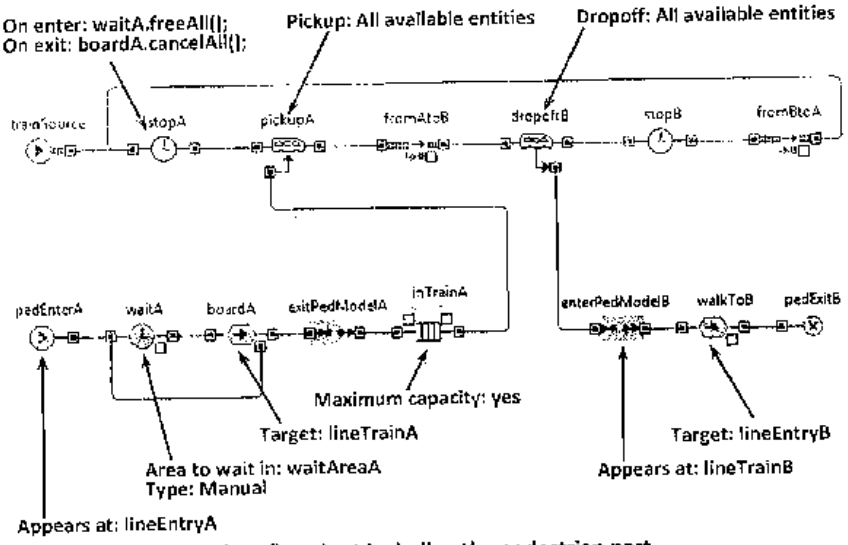


Figure 9.26 The complete flowchart including the pedestrian part

▶ **Add 3D animation:**

13. Drag the **Person** object from the **3D Objects** palette and set its Scale to 50% (remember that 100% scale of people objects correspond to the Pedestrian Library's default 10 pixels per meter scale, and our scale is 5 pixels/meter).
14. Click **pedEnterA** object and set its Animation shape to **person.**
15. Clicck the **track** polyline twice (the first click selects the **groupRails** group and the second – the polyline itself). On the **Advanced** page of its properties set: **Z: -10** and **Z-Height: 1.**
16. Open the **3D** palette and drag **3D Window** to the canvas e.g. below the flowchart. Change its size to, say, 800 x 450.
17. Run the model and view the 3D animation.
18. Draw the platforms. Drag the **Rectangle** object from the **3D** palette, place it at the train stop at terminal A and resize to cover the area where passengers walk to the train. Set its properties:
    Line color: No line
    Fill color: texture concrete
    Z: -2
    Z-Height: 2
19. Ctrl+drag the rectangle to create its copy at the stop at terminal B.
20. Run the model again.

By setting the **z** of the track polyline to -10 and its **z-Height** to 1 we put the rails below the level 0, which by default is the ground level for pedestrians. Correspondingly, we set up the platforms so that their upper surface is at level 0.

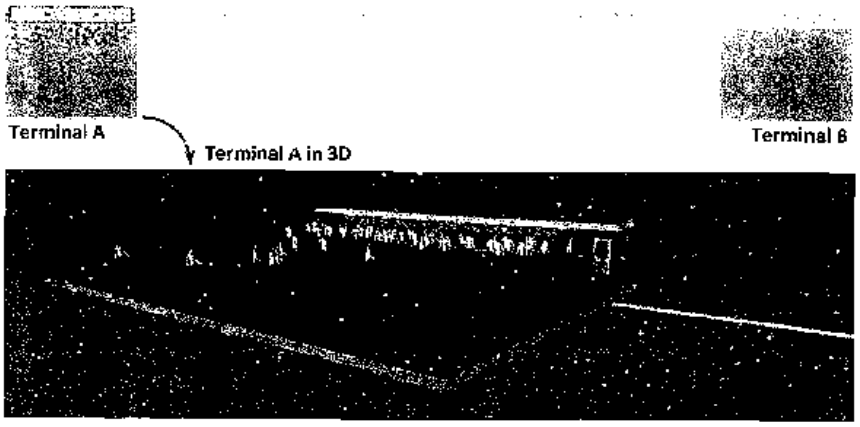


Figure 9.27 2D and 3D animation of the Airport shuttle train

Java class Train (subclass of Entity)

A train is a sequence of one or several rail cars (objects of class **RailCar**) coupled with each other that can move in the rail yard and is controlled by the Rail Library and Enterprise Library flowchart objects.

Trains are created by **TrainSource** objects and must be disposed by **TrainDispose** objects. A train can be split into two trains by **TrainDecouple** and two trains can be combined into one by **TrainCouple**. Trains move in the rail yard under control of **TrainMoveTo** objects. Train class is a subclass of **Entity**, so trains can go through any Enterprise Library objects like **Delay**, **Seize**, **Release**, etc.

The cars in the train are ordered – there will always be the first car and the last car (which are the same in case the train contains only one car), as shown in Figure 9.28. The orientation of the cars at the time of the train creation is the same as that of the train, but later on it can change as a result of coupling/decoupling. The length of the train (the length of the track portion occupied by the train) is equal to the sum of all the car lengths.

The Rail Yard Library does not have a concept of a locomotive, or of any other car type: all rail cars are considered to be equal. Any train can move at any speed, or be coupled/decoupled at any side.

The train has *cruise speed*, *acceleration* and *deceleration* properties. They are initially set up by **TrainSource** but can later on be changed via the Train API. While

the train is moving you can change its speed instantly or by applying acceleration and deceleration.

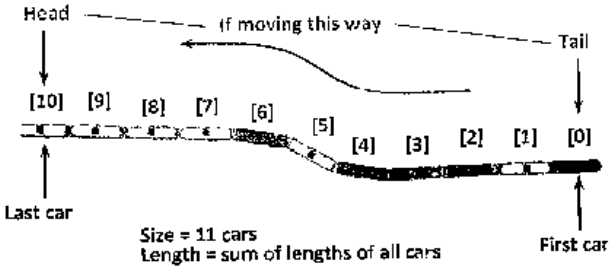


Figure 9.28 A train

Animation of a train is actually the animation of the rail cars of the train, – there are no specific graphics related to the train.

Here are some methods of class **RailCar** (see *Library Reference Guides: Rail Library* (The AnyLogic Company, 2013) for the full list):

Train contents and dimensions

- **int size()** – returns the number of cars in the train
- **double getLength()** – returns the length of the train (the sum of lengths of all cars) in meters
- **RailCar getFirst()** – returns the first car in the train.
- **RailCar getLast()** – returns the last car in the train.
- **RailCar get(int index)** – returns a car with a given index, the first car index is 0, the last car – **size()-1**

Speed and acceleration

- **setCruiseSpeed(double speed)** – sets the cruise speed of the in meters/sec.
- **double getCruiseSpeed()** – returns the cruise speed of the train in meters/sec.
- **setAcceleration(double acceleration)** – sets the acceleration of the train in meters/sec^2.
- **double getAcceleration()** – returns the acceleration of the train in meters/sec^2.
- **setDeceleration(double deceleration)** – sets the deceleration of the train in meters/sec^2.
- **double getDeceleration()** – returns the deceleration of the train in meters/sec^2.

Movement control

- **boolean isMoving()** – returns true if the train is moving, false if not.

- **RailCar getHead()** – returns the car that is at the head of a moving train (either first or last), null if the train is not moving.
- **RailCar getTail()** – returns the car that is at the tail of the moving train (either first or last), null if the train is not moving.
- **setVelocity(double v)** - sets the new velocity of the train. It applies immediately even if the train is moving. If the train is not moving, just remembers the velocity but does not start the train.
- **double getVelocity()** – returns the velocity of the train in meters / sec. If the train is empty, returns 0.
- **accelerateTo(double speed)** – accelerates or decelerates the train to achieve a given speed. Can only be called while the train is moving.
- **ShapePolyLine getTrack(boolean front)** – returns the track (the polyline) where a given side of the train is currently located (if **front** is true, this is front side, otherwise – rear).
- **double getOffset(boolean front)** - returns the offset of a given side of the train relative to the track start point in meters, 0 is the beginning of the track (if **front** is true, this is front side, otherwise – rear).
- **ShapePolyLine getTargetTrack()** – returns the target track of a moving train, null if target is not set.
- **double getTargetOffset()** – returns the target offset (on the target track) of a moving train in meters, assumes the target is set.
- **double getDistanceToTarget()** – returns the distance from the current position to the target point in meters. Assumes the train is moving along a route (specified manually or calculated automatically) and the target is set.

Java class RailCar

In the AnyLogic Rail Library Java class, **RailCar** is the base class for all kinds of rail cars and locomotives. A rail car has dimensions (length and width), can move along a track, and can be coupled and decoupled with another car. A car can have 2D and 3D animation.

Starting with AnyLogic version 6.5.1, rail cars are created by **TrainSource** objects as a part of a train and are fully controlled by trains during their whole lifetime in the rail system. However, the low-level interface to car creation, movement and coupling/decoupling is still supported by the Rail Library for the purpose of compatibility with earlier models.

From the Rail Library viewpoint there is no difference between a car and a locomotive or between freight and passenger cars, but you can make such a distinction in your model. For example, you can specify different dimensions, assign

different animation shapes, or create different subclasses from the **RailCar** class with different properties and methods. And of course, you can associate different behavior logic.

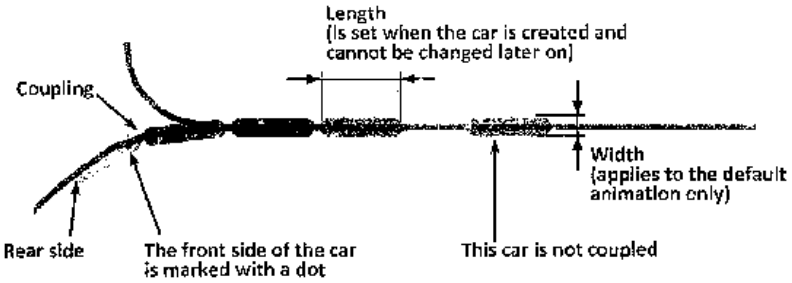


Figure 9.29 Rail car details explained on default 2D animation

A **RailCar** has length and two sides: front and rear. The length of the rail car can only be set up at the time of its creation (e.g. in the parameter **Car lengths** or if you call **setLength()** in the **Car setup** code of **TrainSource**) and cannot be changed later on. You can get the location (track, orientation on track, and offset) of the either side of the car. You can ask the car to *callback* and execute a custom action at the specified point of a given track. Arbitrary information can be passed to the callback code.

The default 2D animation of the car is a rounded rectangle, the front side of which is marked with a black dot as shown in Figure 9.29. If the car is coupled, the connection to the other car is drawn. If the car is highlighted, a circle will be drawn around it. The default 3D animation is a parallelepiped. The color of the default animations can be customized by calling **setColor()**. You can also assign a custom animation shape for a car in the parameter **Car animation** or by calling **setShape()** in the **Car setup** code of **TrainSource**. The 3D Objects palette contains ready-to-use 3D animations for different types of cars and locomotives.

Please note that the size of the rail car custom animation will not be automatically adjusted to the length of the car, so you need to choose the proper scale of the animation shape to ensure there are no spaces between cars in the train and cars are not drawn one above the other. AnyLogic passenger car 3D object at 100% scale is 27 meters long, and other cars and the locomotive are 14 meters long (assuming one pixel is 1 meter).

Here are some methods of class **RailCar** (see *Library Reference Guides: Rail Library* (The AnyLogic Company, 2013) for the full list):

- **setLength(double length)** – sets the length of the car in meters. Can only be done before the car is added to the rail yard.

- **double getLength()** – returns the length of the car in meters.
- **setWidth(double width)** - sets the width of the car in meters. Applies to the default animation only.
- **double getWidth()** – returns the width of the car in meters.
- **setColor(Color color)** – sets the color of the car. Although the color is always stored in a **RailCar** object, it only applies to the default animation.
- **Color getColor()** – returns the color of the car.
- **setShape(Shape shape)** – sets a custom 2D or 3D shape that will be used to animate this car.
- **Shape getShape()** – returns a custom shape that is used to animate the car, or **null**.
- **Train getTrain()** – returns the train the car belongs to, or **null**.
- **Track getTrack(boolean infront)** – returns the track on which a given side of car is currently located (if **infront** is true, this is front side, otherwise rear side).
- **double getOffset(boolean infront)** – returns the offset of a given side of car relative to the track start point in meters: 0 - the beginning of the track (if **infront** is true, this is front side, otherwise rear side).
- **double getX(boolean infront)** – returns the X coordinate of a given side of the car relative to the rail yard group of shapes *in pixels* (if **infront** is true, this is front side, otherwise rear side).
- **double getY(boolean infront)** – same for the Y coordinate.
- **callbackAt(Track track, double offset, Object info)** – requests the car to execute a callback (the code in the field **On at callback** of the **RailYard** object) at the specified point, namely when its side that moves in front reaches a given offset on a given track. You can pass arbitrary information (**Object**) to the callback code to identify what kind of event it is.
- **callbackAt(Track track, ShapeLine line, Object info)** – same as **callbackAt(track, offset, info)** where the offset is the offset of intersection of the track and a given line.

Chapter 10. Java for AnyLogic users

It would be nice if any simulation model could be put together graphically, in drag and drop manner. In practice, however, only very simple models are created by using a mouse and not touching the keyboard. As you try to better reflect the real world in the model, you inevitably realize the need to use probability distributions, evaluate expressions and test conditions containing properties of different objects, define custom data structures and design the corresponding algorithms. These actions are better done in text, not in graphics, and therefore any simulation modeling tool includes a textual scripting language.

From the very beginning we did not want to invent a proprietary scripting language for AnyLogic. Moreover, the creation of AnyLogic was significantly inspired by Java, which we think is the ideal language for modelers. On one hand, Java is a sufficiently high-level language in which you do not need to care about memory allocation, distinguish between objects and references, etc. On the other hand, Java is a fully powerful object oriented programming language with high performance. In Java, you can define and manipulate data structures of any desired complexity; develop efficient algorithms; and use numerous packages available from Sun™, Oracle™ and other vendors. Java is supported by industry leaders and as improvements are made to Java, AnyLogic modelers automatically benefit from it.

A model developed in AnyLogic is fully mapped into Java code and, having been linked with the AnyLogic simulation engine (also written in Java), and, optionally, with a Java optimizer, becomes a completely independent standalone Java application. This makes AnyLogic models cross-platform: they can run on any Java-enabled environment or even in a web browser as applets.

A frequently asked question is "How much Java do I need to know to be successful with AnyLogic?" The good news is that you do not need to learn object-oriented programming. The "backbone Java class structure" of the model is automatically generated by AnyLogic. In a typical model, Java code is present in small portions written in various properties of the graphically-created model objects. This can be an expression, a function call, or a couple of statements. Therefore you need to get familiar with the fundamental data types, learn the basics of Java syntax, and understand that to do something with a model object you need to call its function.

This chapter is by no means a complete description of Java language, or even an introduction to Java suitable for programmers. This is a collection of information that will allow you to manipulate data and model objects in AnyLogic models. It is sufficient for a typical modeler. For those who plan to write sophisticated Java code,

use object-orientedness, or work with external Java packages, we recommend learning Java with a good textbook, such as this one:

10.1. Primitive data types

There are about ten *primitive data types* in Java. In AnyLogic models we typically use these four:

Type name	Represents	Examples of constants
int	Integer numbers	12 10000 -15 0
double	Real numbers	877.13 12.0 12. 0.153 .153 -11.7 3.6e-5
boolean	Boolean values	true false
String	Text strings	"AnyLogic" "X = " "Line\nNew line" ""

The word "*double*" means real value with double precision. In the AnyLogic engine all real values (such as time, coordinates, length, speed, and random numbers) have double precision. The type String is actually a class (a non-primitive type, notice that its name starts with a capital letter), but it is a fundamental class, so some operations with strings are built into the core of Java language.

Consider the *numeric constants*. Depending on the way you write a number, Java will treat it either as real or as integer. Any number with the decimal delimiter "." is treated as a real number, even if its fractional part is missing or contains only zeros (this is important for integer division, see Section 10.5). If either the integer or fractional part is zero, it can be skipped, so ".153" is the same as "0.153", and "12." is the same as "12.0".

Boolean constants in Java are true and false and, unlike in languages such as C or C++, they are not interchangeable with numbers, so you cannot treat false as 0 or a non-zero number as true.

String constants are sequences of characters enclosed between the quotation marks. The empty string (the string containing no characters) is denoted as "". Special characters are included in string constants with the help of *escape sequences* that start with the backslash. For example, end of line is denoted by \n, so the string "Line one\nLine two" will appear as:

```
Line one
Line two
```

If you wish to include the quote character in a string, you need to write \". For example the string constant "String with \" in the middle" will print as:

```
String with " in the middle.
```

To include a backslash in a string, use a double backlash. For example, "This is a backslash: \\" will print as:

```
This is a backslash: \
```

10.2. Classes

Structures more complex than primitive types are defined with the help of classes and in object-oriented manner. The mission of explaining the concepts of object-oriented design is impossible within the scope of this chapter: the subject deserves a separate book, and there are a lot of them already written. All we can do here is give you a feeling of what object-orientedness is about by introducing such fundamental terms as class, method, object, instance, subclass, and inheritance, and show how Java supports object-oriented design.

You should not try to learn or fully understand the code fragments in this section. It will be sufficient if, having read this section, you will know that, for example, a **statechart** in your model is an instance of AnyLogic class **Statechart** and you can find out its current state by calling its method: **statechart.getActiveSimpleState()**, or if the class **Agent** is a subclass of **ActiveObject**, it supports the methods of both of them.

Class as grouping of data and methods. Objects as instances of class

Consider an example. Suppose you are working a with local map and use coordinates of locations and calculate distances. Of course you can remember two double values x and y for each location and write a function **distance(x1, y1, x2, y2)** that would calculate distances between two pairs of coordinates. But it is a lot more elegant to introduce a new entity that would group together the coordinates and the method of calculating distance to another location. In object-oriented programming such an entity is called a *class*. Java class definition for the location on a local map can look like this:

```
class Location {

  //constructor: creates a Location object with given coordinates
  Location( double xcoord, double ycoord ) {
    x = xcoord;
    y = ycoord;
  }

  //two fields of type double
  double x; //x coordinate of the location
  double y; //y coordinate of the location
```

```
  //method (function): calculates distance from this location to another one
  double distanceTo( Location other ) {
    double dx = other.x - x;
    double dy = other.y - y;
    return sqrt( dx*dx + dy*dy );
  }

}
```

As you can see, a class combines data and methods that work with the data.

Having defined such a class, we can write very simple and readable code when working with the map. Consider the following code:

```
Location origin = new Location( 0, 0 ); //create first location
Location destination = new Location( 250, 470 ); //create second location
double distance = origin.distanceTo( destination ); //calculate distance
```

The locations **origin** and **destination** are *objects* and are *instances* of the class **Location**. The expression **new Location(250, 470)** is a *constructor call*; it creates and returns a new instance of the class **Location** with the given coordinates. The expression **origin.distanceTo(destination)** is a *method call*; it asks the object **origin** to calculate the distance to another object **destination**.

If you declare a variable of a non-primitive type (of a class) and do not initialize it, its value will be set to **null** (**null** is a special Java literal that denotes "*nothing*"). Sometimes you explicitly assign **null** to a variable to "forget" the object it referred to and to indicate that the object is missing or unavailable.

```
Location target; //a variable is declared without initialization. target equals null
target = warehouse; //assign the object (pointed to by the variable) warehouse to target
//now target and warehouse point to the same object
...
target = null; //target forgets about the warehouse and equals null again
```

Inheritance. Subclass and super class

Now suppose some locations in your 2D space correspond to cities. A city has a name and population size. To efficiently manipulate cities we can extend the class **Location** by adding two more fields: **name** and **population**. We will call the new entity **City**. In Java this will look like:

```
class City extends Location { //declaration of the class City that extends the class Location

  //constructor of class City
  City( String n, double x, double y ) {
    super( x, y ); //call of constructor of the super class with parameters x and y
    name = n;
```

```
    //the population field is not initialized in the constructor – to be set later
  }

  //fields of class City
  String name;
  int population;

}
```

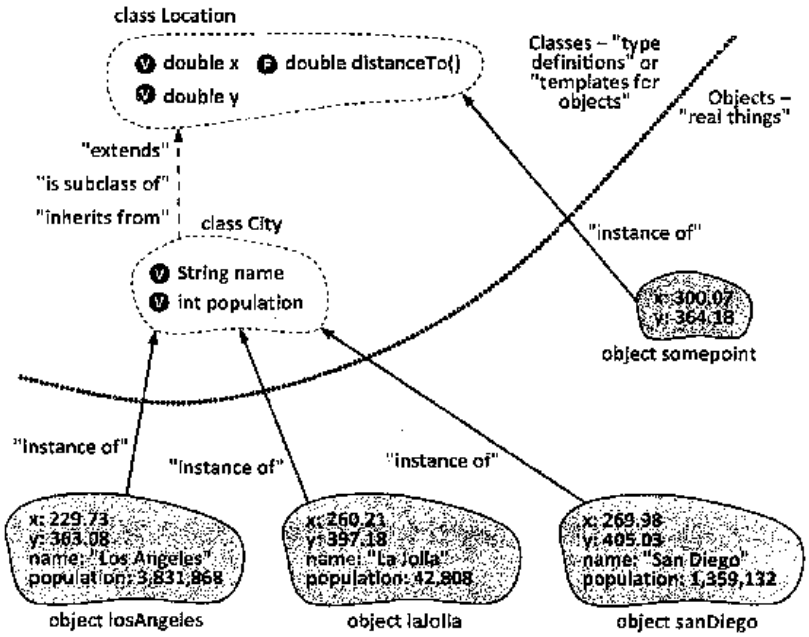


Figure 10.1 Classes and Inheritance

City is called a *subclass* of Location, and Location is correspondingly a *super class* (or *base class*) of City. City *inherits* all properties of Location and adds new ones. See how elegant the code is for finding the biggest city in the range of 100 kilometers from a given point of class Location (we assume that there is a collection cities where all cities are included):

```
int pop = 0; //here we will remember the largest population found so far
City biggestcity = null; //here we will store the best city found so far

for( City city : cities ) { //for each city in the collection cities
  if( point.distanceTo( city ) < 100 && city.population > pop ) { //if best so far
    biggestcity = city; //remember it
    pop = city.population; //and remember its population size
  }
```

```
}
traceln( "The biggest city within 100km is " + city.name ); //print the search result
```

Notice that although city is an object of class City, which is "bigger" than Location, it can still be treated as Location when needed, in particular when calling the method distanceTo() of class Location.

This is a general rule: an object of a subclass can always be considered as an object of its base class.

How about vice versa? You can declare a variable of class Location and assign it an object of class City (a subclass of Location):

```
Location place = lajolla;
```

This will work. However, if you then try to access the population of place, Java will signal an error:

```
int p = place.population; //error: "population cannot be resolved or is not a field"
```

This happens because Java does not know that place is in fact an object of class City. To handle such situations you can:

- test whether an object of a certain class is actually an object of its particular subclass by using the operator instanceof: *<object>* instanceof *<class>*
- "*cast*" the object from a super class to a subclass by writing the name of the subclass in parentheses before the object: (*<class>*)*<object>*

A typical code pattern is:

```
if( place instanceof City ) {
  City city = (City)place;
  int p = city.population;
  ...
}
```

Classes and objects in AnyLogic models

Virtually all objects of which you create your models are instances of AnyLogic Java classes. In the Table you will find the Java class names for most model elements you work with. Whenever you need to find out what can you do with a particular object using Java, you should find the corresponding Java class in *AnyLogic Classes and Functions* (The AnyLogic Company, 2013) and look through its methods and fields.

Model object	Java class name
Active object (such as Main)	Base class: **ActiveObject**
Agent (such as **Person**)	Base class: **Agent** (subclass of **ActiveObject**) Classes: **AgentContinuous2D, AgentContinuous3D, AgentDiscrete2D, AgentContinuousGIS**
Event	Base class: **Event** Classes: **EventCondition, EventRate, EventTimeout**
Dynamic event	**DynamicEvent**
Statechart	**Statechart**
Transition of a statechart	Base class: **Transition** Classes: **TransitionCondition, TransitionMessage, TransitionRate, TransitionTimeout**
Table function	**TableFunction**
Schedule	**Schedule**
Port	Base class: **Port** Classes of the Enterprise Library: **InPort, OutPort, OutPortPush**
Statistic collectors: Data set, Histogram Data, ...	**DataSet, HistogramData, ...**
Charts: Bar chart, Plot, Histogram chart, ...	**BarChart, Plot, Histogram, ...**
Shapes: Rectangle, Polyline, Group, ...	Base class: **Shape** Classes: **ShapeRectangle, ShapePolyLine, ShapeGroup, ...**
Controls: Button, Slider, List box, ...	Base class: **ShapeControl** (subclass of **Shape**) Classes: **ShapeButton, ShapeSlider, ShapeListBox**
Connectivity objects: Text file, Excel file, Database, ...	**TextFile, ExcelFile, Database, ...**
Enterprise Library objects: Source, Delay, Queue, Select Output, ...	Base class: **ActiveObject** Classes of the Enterprise Library: **Source, Queue, Delay, SelectOutput, ...**
Entities and resource units in process models	Classes of the Enterprise Library: **Entity, ResourceUnit**
Experiments: Simulation, Parameter variation, Optimization, ...	Base class: **Experiment** Classes: **ExperimentSimulation, ExperimentParamVariation, ExperimentOptimization**

AnyLogic model GUI	**Presentation, Panel**

10.3. Variables (local variables and class fields)

In this section we are considering Java variables. Special kinds of variables with additional functionality specific to simulation modeling, such as parameters or dynamic variables are described in other chapters of the book.

Depending on where a variable is declared it can be either a:

- *Local variable* – an auxiliary temporary variable that exists only while a particular function or a block of statements is executed, or
- *Class variable* (or *class field* – more correct term in Java) – a variable that is present in any object of a class, and whose lifetime is the same as the object lifetime.

Local (temporary) variables

Local variables are declared in sections of Java code such as a block, a loop statement, or a function body. They are created and initialized when the code section execution starts and disappear when it finishes. The declaration consists of the variable type, name, and optional initialization. The declaration is a statement (see Section 10.8), so it should end with a semicolon. For example:

```
double sum = 0; //double variable sum initially 0
int k; //integer variable k, not initialized
String msg = ok ? "OK" : "Not OK"; //string variable msg initialized with expression
```

Local variables can be declared and used in AnyLogic fields where you enter actions (sequences of statements), such as **Startup code** of the active object class, **Action** field of events or transitions, **Entry action** and **Exit action** of state, **On enter** and **On exit** fields of flowchart objects. In Figure 10.2 the variables **sum** and **p** are declared in the action code of the event **endOfYear** and exist only while this portion of code is being executed.

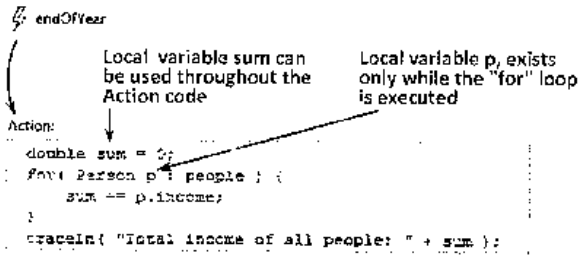

endOfYear
Local variable sum can be used throughout the Action code
Local variable p, exists only while the "for" loop is executed
Action:
double sum = 0;
for(Person p : people) {
sum += p.income;
}
traceln("Total income of all people: " + sum);

Figure 10.2 Local variables declared in the Action code of an event

Class variables (fields)

Java variables (fields) of the active object class are part of the "memory" or "state" of active objects. They can be declared graphically or in code.

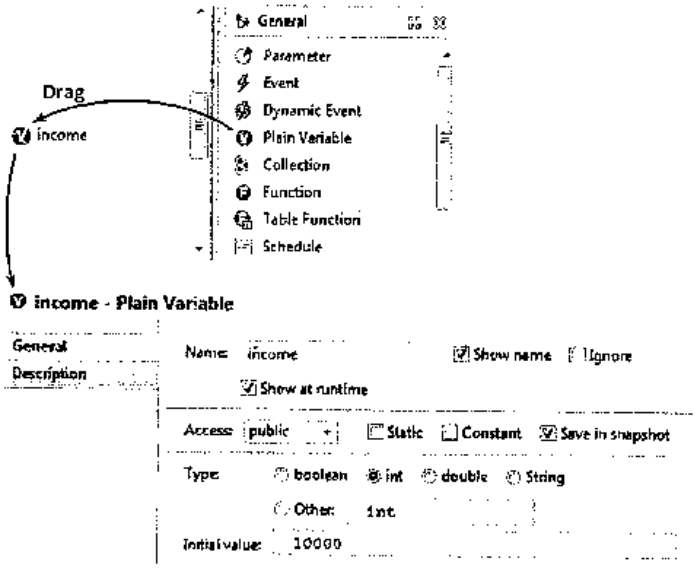


Figure 10.3 A variable of an active object declared in the graphical editor

▶ **To declare a variable of active object (or experiment) class:**

1. Open the **General** palette and drag the **Variable** object to the canvas.
2. Type the variable name in the in-place editor or in the variable properties.
3. Choose the variable **Access** type in the **General** page of the variable properties. In most cases you can leave **default**, which means the variable

will be visible within the current model. **public** opens access to the variable from other models, and **private** limits access to this active object only.

4. Choose the variable type. If the type is not one of the primitive types, you should choose **Other** and enter the type name in the field nearby.
5. Optionally you can enter the variable Initial value.

If you do not specify the initial value, it will be **false** for **boolean** type, **0** for numeric variables, and **null** ("nothing") for all other classes including **String**.

In Figure 10.4, a variable **income** of type **int** is declared in an active object (or experiment) class. Its access type is **public**, therefore it will be accessible from anywhere. The initial value of the variable is **10000**. The graphical declaration above is equivalent to a line of code of the class, which you can write in the **Additional class code** field of the **Advanced** property page of the class, as shown in Figure 10.4:

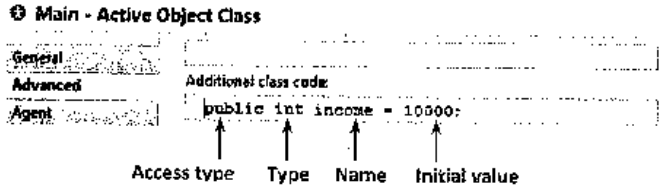


Figure 10.4 The same variable declared in the Additional class code field

Graphical declaration of a variable allows you to visually group it together with related functions or objects, and to view or change the variable value at runtime with one click.

10.4. Functions (methods)

In Java, to call a *function* (or *method*, which is more correct in object-oriented languages like Java, as any function is a method of a class) you write the function name followed by the argument values in parentheses. For example, this is a call of a triangular probability distribution function with three numeric arguments:

```
triangular( 2, 5, 14 )
```

The next function call prints the coordinates of an agent to the model log with a timestamp:

```
traceln( time() + ": X = " + getX() + " Y = " + getY() );
```

The argument of this function call is a string expression with five components; three of them are also function calls: **time()**, **getX()**, and **getY()**.

Even if a function has no arguments, you must put parentheses after the function name, like this: **time()**

A function may or may not return a value. For example, the call of **time()** returns the current model time of type **double**, and the call of **traceln()** does not return a value. If a function returns a value, it can be used in an expression (like **time()** was used in the argument expression of **traceln()**). If a function does not return a value it can only be called as a statement (the semicolon after the call of **traceln()** indicates that this is a statement, see Section 10.8).

Standard and system functions

Most of the code you write in AnyLogic is the code of a subclass of **ActiveObject** (a fundamental class of the AnyLogic simulation engine). For your convenience, AnyLogic system functions and most frequently used Java standard functions are available directly through **ActiveObject** (you do not need to think which package or class they belong to, and can call those functions without any prefixes). Following are some examples (these are only a few functions out of several hundreds; see *AnyLogic Classes and Functions* (The AnyLogic Company, 2013) for the full list).

The type name written before the function name indicates the type of the returned value. If the function does not return a value, we write **void** instead of type, but we are dropping it here.

Mathematical functions (imported from **java.lang.Math**, about 45 functions in total):

- **double min(a, b)** – returns the minimum of **a** and **b**
- **double log(a)** – returns the natural logarithm of **a**
- **double pow(a, b)** – returns the value of **a** raised to the power of **b**
- **double sqrt(a)** – returns the square root of **a**

Functions related to the model time, date, or date elements (about 20 functions):

- **double time()** – returns the current model time (in model time units)
- **Date date()** – returns the current model date (Date is standard Java class)
- **int getMinute()** – returns the minute within the hour of the current model date
- **double minute()** – returns one minute time interval value in the model time units

Probability distributions (over 30 distributions are supported):

- **double uniform(min, max)** – returns a uniformly distributed random number
- **double exponential(rate)** – returns an exponentially distributed random number

Output to the model log and formatting:

- traceln(Object o) – prints a string representation of an object with a line delimiter at the end to the model log
- String format(value) – formats a value into a well-formed string

Model execution control:

- boolean finishSimulation() – causes the simulation engine to terminate the model execution after completing the current event.
- boolean pauseSimulation() – puts the simulation engine into a "paused" state.
- error(String msg) – signals an error. Terminates the model execution with a given message.

Navigation in the model structure and the execution environment:

- ActiveObject getOwner() – returns the upper level active object that embeds this one, if any
- int getIndex() – returns the index of this active object in the list if it is a replicated object
- Experiment<?> getExperiment() – returns the experiment controlling the model execution
- Presentation getPresentation() – returns the model GUI
- Engine getEngine() – returns the simulation engine

If the active object is an agent, more functions are available specific to the particular type of agent, for example:

Network and communication-related functions:

- connectTo(agent) – establishes a connection with another agent
- send(msg, agent) – sends a message to given agent

Space and movement-related functions:

- double getX() – returns the X-coordinate of the agent in continuous space
- moveTo(x, y, z) – starts movement of the agent to the point (x,y,z) in 3D space

The functions available in the current context (for example, in a property field where you are entering code), are always listed in the *code completion* window that opens if you press Ctrl+Space (Alt+Space on Mac), as shown in Figure 10.5.

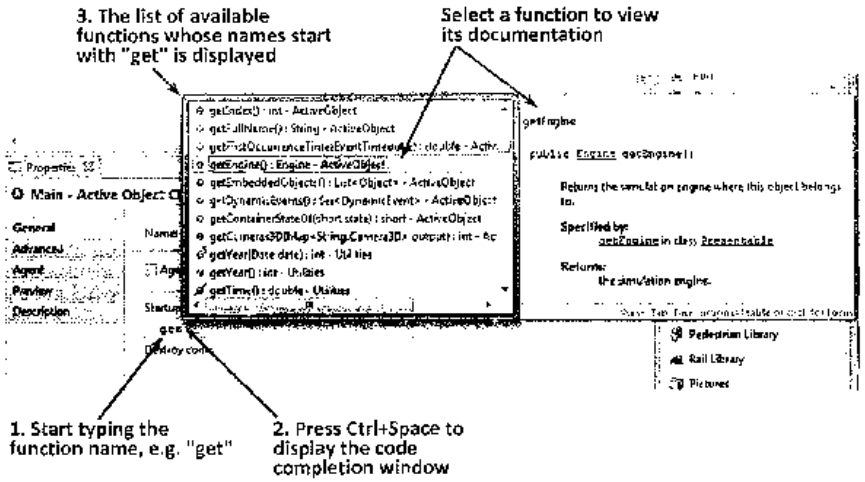

3. The list of available functions whose names start with "get" is displayed
Select a function to view its documentation
1. Start typing the function name, e.g. "get"
2. Press Ctrl+Space to display the code completion window

Figure 10.5 Code completion window shows the functions available in the current context

Functions of the model elements

All elements in AnyLogic model (events, statecharts, table functions, plots, graphics shapes, controls, library objects, and so on) are mapped to Java objects and expose Java *API* (Application Programming Interface) to the user. You can retrieve information about the objects and control them using their API.

To call a function of a particular model element that is inside the active object you should put the element name followed by dot "." before the function call:
<object>.<method call>

Following are some examples of calling functions of the elements of the current active object (the full list of functions for a particular element is available in *AnyLogic Help* (The AnyLogic Company, 2013):

Scheduling and resetting events:

- **event.restart(15*minute())** – schedules the event to occur in 15 minutes
- **event.reset()** – resets a (possibly scheduled) event

Sending messages to statecharts and obtaining their current states:

- **statechart.receiveMessage("Go!")** – delivers the message "Go!" to the statechart
- **statechart.isStateActive(going)** – tests if the state going is currently active in the statechart

Adding a sample data point to a histogram:

- **histData.add(x)** – adds the value of x to the histogram data object **histData**

Display a view area:

- **viewArea.navigateTo()** – displays the part of the canvas marked by the **viewArea**

Changing the color of a shape:

- **rectangle.setFillColor(red)** – sets the fill color of the **rectangle** shape to red

Retrieving the current value of a checkbox:

- **boolean checkbox.isSelected()** – returns the current state of the **checkbox**

This statement hides or shows the shape depending on the state of the checkbox:

- **rectangle.setVisible(checkbox.isSelected());**

Changing parameters and states of embedded active objects:

- **source.set_rate(100)** – sets the **rate** parameter of **source** object to 100
- **hold.setBlocked(true)** – puts the **block** object to the blocked state

Note that the parameter **rate** appears as **Arrival rate** on the **General** page of the **source** object properties. To find out the Java names of the parameters, you should open the **Parameters** page of the properties.

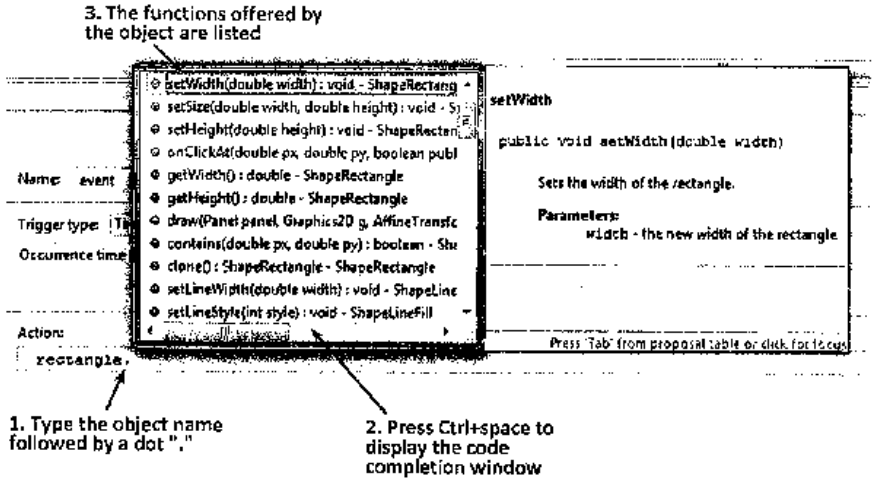

3. The functions offered by the object are listed
setWidth(double width) : void - ShapeRectang
setSize(double width, double height) : void - S
setHeight(double height) : void - ShapeRecten
onClickAt(double px, double py, boolean publ
getWidth() : double - ShapeRectangle
getHeight() : double - ShapeRectangle
draw(Panel panel, Graphics2D g, AffineTransfc
contains(double px, double py) : boolean - Sha
clone() : ShapeRectangle - ShapeRectangle
setLineWidth(double width) : void - ShapeLine
setLineStyle(int style) : void - ShapeLineFill
setWidth
public void setWidth(double width)
Sets the width of the rectangle.
Parameters:
width - the new width of the rectangle
Press 'Tab' from proposal table or click for focus
Name: event
Trigger type:
Occurrence time:
Action:
rectangle.
1. Type the object name followed by a dot "."
2. Press Ctrl+space to display the code completion window

Figure 10.6 Code completion widow showing the functions of a particular object

You can easily find out which functions are offered by a particular object by using code completion. In the code you are writing, you should type the object name, then dot ".", and then press Ctrl+Space (Alt+Space on Mac). The pop-up window will display the list of functions you can call.

Typically, the list of suggested functions in the code completion window is large, and it makes sense to apply a filter. For example, if you are looking for a function that changes a property of the object, you should type the object name, dot, and "set". The completion suggestions will then be limited to those that start with "set". See Section 10.7 for more tips.

Defining your own function

You can define your own functions of active objects, experiments, and custom Java classes. For active objects and experiments, functions can be defined as objects in the graphical editor.

▶ **To define a function of an active object (or experiment) class:**

1. Open the **General** palette and drag the **Function** object to the canvas.
2. Type the function name in the in-place editor or in the function properties.
3. Choose the function **Access** type in the **General** page of the function properties. In most cases you can leave **default**, which means the function will be visible within the current model. **public** opens the access to the function from other models, and **private** limits the access to this active object only.
4. Choose the function return type. **void** means the function does not return a value. If the return type is not one of the primitive types, you should choose **Other** and enter the type name in the field nearby.
5. If the function has arguments, add them to the **Function arguments** table. You must specify the name and type for each argument. Use the buttons to the right of the table to reorder or delete arguments.
6. Switch to the **Code** page of the function properties and type the function body code (the page may have some auto-generated code for functions returning a value).

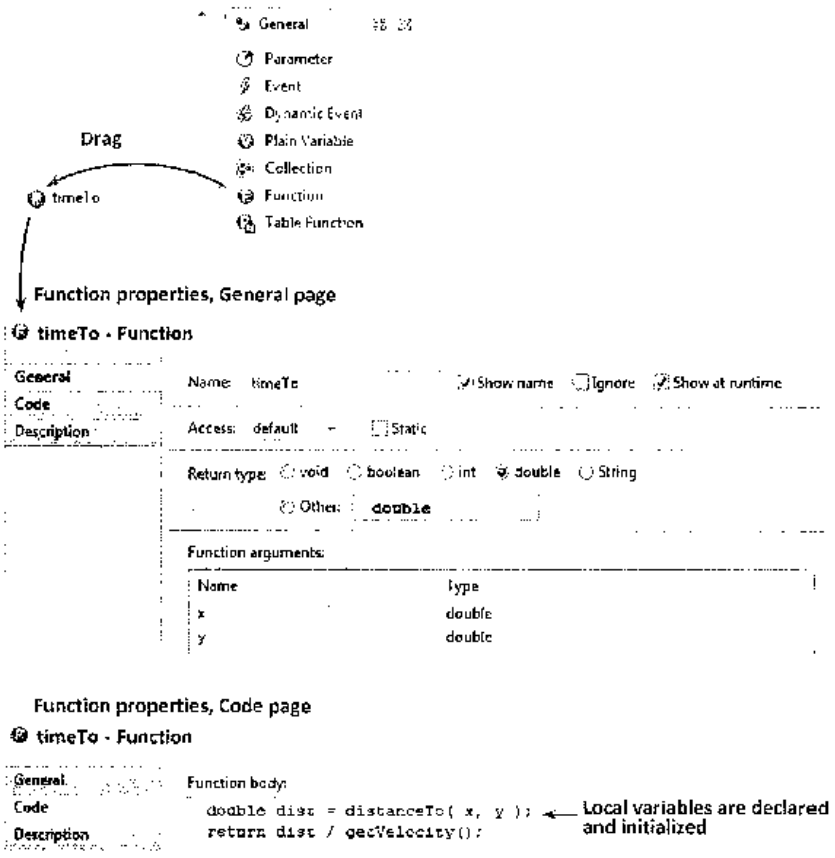


Figure 10.7 A user-defined function of an active object

In Figure 10.7, the function **timeTo()** is defined for an active object class, which is an agent in continuous 2D space. The function calculates and returns the time needed for the agent to move to a given point (**x**,**y**) in the 2D space. The returned value is of type **double**, and the two arguments **x** and **y** are of type **double** as well. In the body code of the function, we declare a local variable **dist** and initialize it with the distance to the point. Then we divide the distance by the velocity of the agent and return the result.

When AnyLogic generates the Java code for the active object class, the function will be mapped to the following fragment of the class code (you can view the code by placing the cursor in the function body and pressing Ctrl+J):

```
double timeTo( double x, double y ) {
  double dist = distanceTo( x, y );
  return dist / getVelocity();
}
```

Another way of defining a function is writing its full Java declaration and implementation in the **Additional class code** field of the **Advanced** property page of the active object class or experiment, as shown in Figure 10.8. The two definitions of the functions are absolutely equivalent, but having the function icon on the canvas is preferred because it is more visual and provides quicker access to the function code in design time.

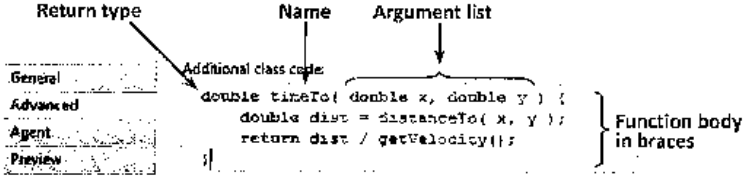


Figure 10.8 The same function defined in the Additional class code of the active object

10.5. Expressions

Arithmetic expressions

Arithmetic expressions in Java are composed with the usual operators +, –, *, / and the remainder operator %. Multiplication and division operations have higher priority than addition and subtraction. Operations with equal priority are performed from left to right. Parentheses are used to control the order of operation execution.

```
a+b/c≡a+(b/c)
a*b−c≡(a*b)−c
a/b/c≡(a/b)/c
```

It is recommended to always use parentheses to explicitly define the order of operations, so you do not have to remember which operation has higher priority.

A discussion on arithmetic expressions should include *integer division*.

The result of division in Java depends on the types of the operands. If both operands are integers, the result will be an integer as well. The unintended use of integer division may therefore lead to significant precision loss. To let Java perform real division (and get a real number as a result) at least one of the operands must be of real type.

For example:

3 / 2 ≡ 1

2 / 3 ≡ 0

because this is integer division. However,

3 / 2. ≡ 1.5

2.0 / 3 ≡ 0.66666666...

because **2.** and **2.0** are real numbers. If **k** and **n** are variables of type **int**, **k/n** is integer division. To perform a real division over two integer variables or expressions you should force Java to treat at least one of them as real. This is done by *type casting*. You need to write the name of the type you are converting to before the variable in parentheses. For example, **(double)k/n** will be real division with result of type **double**.

Integer division is frequently used together with the *remainder operation* to obtain the row and column of the item from its sequential index. Suppose you have a collection of 600 items, say, seats in a theater, and want to arrange them in 20 rows, each row containing 30 seats. The expressions for the seat number in a row and the row would be:

Seat number: **index % 30** (remainder of division of index by 30: 0 - 29)

Row number: **index / 30** (integer division of index by 30: 0 - 19)

where index is between 0 and 599. For example, the seat with index 247 will be in row 8 with seat number 7. In Example 12.6: "Selling seats in the movie theater" this technique is used to position the replicated picture of a seat.

The *power operation* in Java does not have an operand (if you write **a^b** this will mean bitwise OR and not power). To perform the power operation you need to call the **pow()** function:

pow(a, b) ≡ a^b

Java supports several useful shortcuts for frequent arithmetic operations over numeric variables. They are:

i++ ≡ i = i+1 (increment **i** by 1)

i— ≡ i = i–1 (decrement **i** by 1)

a += 100.0 ≡ a = a + 100.0 (increase **a** by 100.0)

b –= 14 ≡ b = b – 14 (decrease **b** by 14)

Note that, although these shortcuts can be used in expressions, their evaluation has effect: it changes the value of the operands.

Relations and equality

Relations between two numeric expressions are determined using the following operators:

- `>` greater than
- `>=` greater than or equal to
- `<` less than
- `<=` less than or equal to

You can test if two operands (primitive or objects) are equal using the two operators:

- `==` equal to
- `!=` not equal to

For non-primitive objects (i.e. for those that are not numeric or **boolean**) the operators "==" and "!=" test if the two operands are *the same object* rather than *two objects with equal contents*. To compare the contents of two objects, e.g. two strings, you should use the function **equals()**.

For example, to test if the string message **msg** equals "**Wake up!**" you should write:

```
msg.equals( "Wake up!" )
```

Do not confuse the equality operator "==" with the assignment operator "="!

`a = 5` means assign value of 5 to the variable **a**, whereas

`a == 5` is true if a equals 5 and false otherwise

The result of all comparison operations is of **boolean** type (**true** or **false**).

Logical expressions

There are three logical operators in Java that can be applied to **boolean** expressions:

- `&&` logical AND
- `||` logical OR
- `!` logical NOT (unary operator)

AND has higher priority than OR, so that

```
a || b && c = a || ( b && c )
```

but again, it is always better to put parentheses to explicitly define the order of operations.

The logical operations in Java exhibit so-called *short-circuiting behavior*, which means that the second operand is evaluated only if needed.

This feature is very useful when a part of an expression cannot be evaluated (will cause an error) if another part is not true. For example, let destinations be a collection of places an agent in the model must visit. To test if the first place to visit is London, you can write:

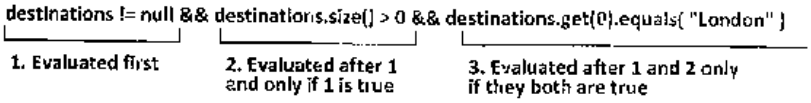

destinations != null && destinations.size() > 0 && destinations.get(0).equals("London")
1. Evaluated first
2. Evaluated after 1 and only if 1 is true
3. Evaluated after 1 and 2 only if they both are true

Figure 10.9 Short-circuiting behavior of logical operations

Here we first test if the list of destinations exists at all (does not equal null), then, if it exists, we test if it has at least one element (its size is greater than 0), and if true, we compare that element with the string "London".

String expressions

Strings in Java can be concatenated by using the "+" operator. For example:

"Any" + "Logic" results in "AnyLogic"

Moreover, this way you can compose strings from objects of different types. The non-string objects will be converted to strings and then all strings will be concatenated into one. This is widely used in AnyLogic models to display the textual information on variable values. For example, in the dynamic property field Text of the Text object, you can write:

```
"x = " + x
```

And then at runtime the text object will display the current value of x in the textual form, e.g.:

x = 14.387

You can use an empty string in such expressions as well: "" + x will simply convert x to string. Another example is the following string expression:

```
"Number of destinations: " + destinations.size() + "; the first one is " + destinations.get(0)
```

which results in a string like this:

Number of destinations: 18; the first one is London

And once again: you should use the function equals() and not the "==" operator to compare strings!

Conditional operator ?:

Conditional operator is helpful when you need to use one of the two different values in an expression depending on a condition. It is a ternary operator, i.e. it has three operands:

<condition> **?** *<value if true>* **:** *<value if false>*

It can be applied to values of any type: numeric, boolean, strings, or any classes. The following expression returns 0 if the backlog contains no orders or returns the amount of the first order in the backlog queue:

```
backlog.isEmpty() ? 0 : backlog.getFirst().amount
```

Conditional operators can be nested. For example, the following code prints the level of income of a person (High, Medium, or Low) depending on the value of the variable income:

```
traceln( "Income: " + ( income > 10000 ? "High" : ( income < 1500 ? "Low" : "Medium" ) ) );
```

This single code line is equivalent to the following combination of "if" statements:

```
trace( "Income: " );
if( income > 10000 ) {
  traceln( "High" );
} else if( income < 1500 ) {
  traceln( "Low" );
} else {
  traceln( "Medium" );
}
```

10.6. Java arrays and collections

Java offers two types of constructs where you can store multiple values or objects of the same type: arrays and collections (for System Dynamics models AnyLogic also offers **HyperArray**, also known as "subscripts", – a special type of collection for dynamic variables; it is described in Chapter 5, "System dynamics and dynamic systems").

Array or collection? Arrays are simple constructs with linear storage of fixed size and therefore they can only store a given number of elements. Arrays are built into the core of Java language and the array-related Java syntax is very easy and straightforward. For example the n^{th} element of the **array** can be obtained as **array[n]**. Collections are more sophisticated and flexible. First of all, they are resizable; you can add any number of elements to a collection. A collection will automatically handle deletion of an element from any position. There are several types of collections with different internal storage structure (linear, list, hash set, tree, etc.) and you can choose a collection type best matching your problem so that your most

frequent operations will be convenient and efficient. Collections are Java classes. For example, the syntax for obtaining the n^{th} element of a collection of type ArrayList is collection.get(n).

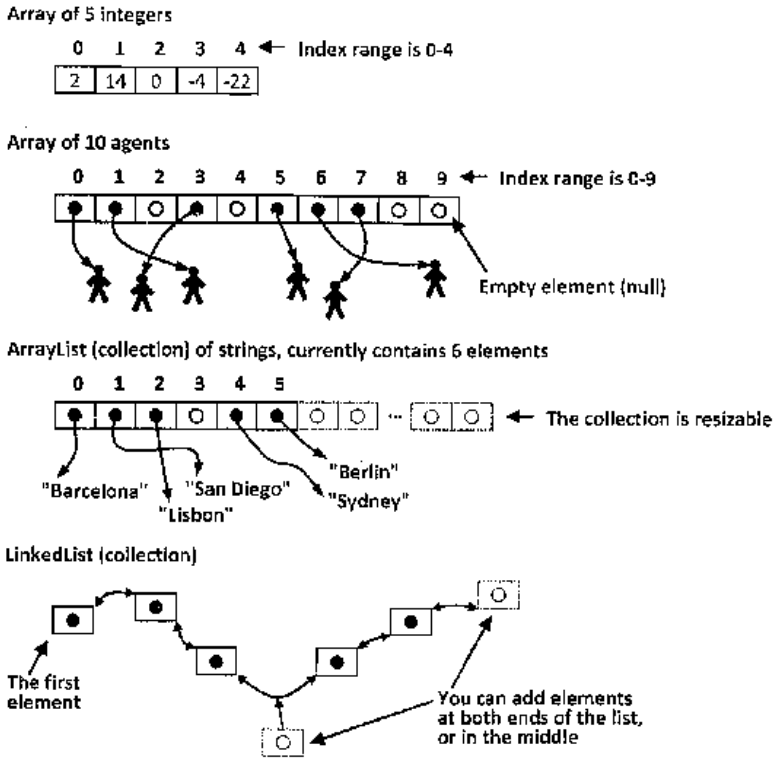

Array of 5 integers
0 1 2 3 4
Index range is 0-4
2 14 0 -4 -22
Array of 10 agents
0 1 2 3 4 5 6 7 8 9
Index range is 0-9
Empty element (null)
ArrayList (collection) of strings, currently contains 6 elements
0 1 2 3 4 5
The collection is resizable
"Barcelona"
"San Diego"
"Lisbon"
"Berlin"
"Sydney"
LinkedList (collection)
The first element
You can add elements at both ends of the list, or in the middle

Figure 10.10 Java arrays and collections

Please note that indexes in Java arrays and collections start with 0, not with 1! In an array or collection of size 10 the index of the first element is 0, and the last element has index 9.

Arrays

Java arrays are containers with linear storage of fixed size. To create an array you should declare a variable of array type and initialize it with a new array, like this:

```
int[] intarray = new int[100]; //array of 100 integer numbers
```

Array type is composed of the element type followed by square brackets, e.g. **int[]**, **double[]**, **String[]**, or **Agent[]**. The size of the array is not a part of its type. Allocation of

the actual storage space (memory) for the array elements is done by the expression **new int[100]**, and this is where the size is defined. Note that unless you initialize the array with the allocated storage, it will be equal to **null**, and you will not be able to access its elements.

A graphical declaration of the same array of 100 integers will look like Figure 10.11. You should use a **Variable** or **Parameter** object, choose **Other** for **Type** and enter the array type in the field on the right.

Do not be confused by the checkbox **Array** in the properties of **Parameter**: that checkbox sets the type of parameter to System Dynamics **HyperArray** and not to Java array.

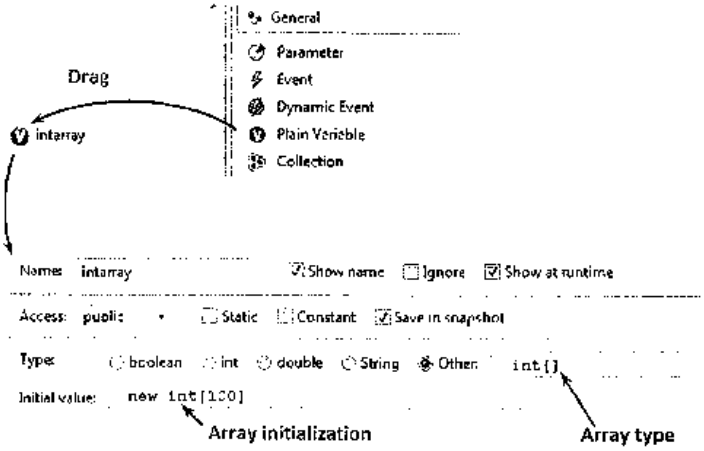


Figure 10.11 Array variable declared graphically

All elements of the array initialized that way will be set to **0** (for numeric element type), **false** for **boolean** type and **null** for all classes including **String**. Another option is to explicitly provide initial values for all array elements. The syntax is the following:

```
int[] intarray = new int[] { 13, x-3, -15, 0, max( a, 100 ) };
```

The size of the array is then defined by the number of expressions in braces. To obtain the size of an existing array you should write *<array name>*.length, for example:

```
intarray.length
```

The i^{th} element of an array can be accessed as:

```
intarray[i]
```

Iteration over array elements is done by index. The following loop increments each element of the array (remember that array indexes are 0-based):

```
for( int i=0; i<intarray.length; i++ ) {
  intarray[i]++;
}
```

java also supports a simpler form of the "for" loop for arrays. The following code calculates the sum of the array elements:

```
int sum = 0;
for( int element : intarray ) {
  sum += element;
}
```

Arrays can be *multidimensional*. This piece of code creates a two-dimensional array of double values and initializes it in a loop:

```
double[][] doubleArray = new double[10][20];
for( int i=0; i<doubleArray.length; i++ ) {
  for( int j=0; j<doubleArray[i].length; j++ ) {
    doubleArray[i][j] = i * j;
  }
}
```

You can think of a multidimensional array as an "array of arrays". The array initialized as new **double[10][20]** is an array of 10 arrays of 20 double values each. Notice that **doubleArray.length** returns 10 and **doubleArray[i].length** returns 20.

Collections

Collections are Java classes developed to efficiently store multiple elements of a certain type. Unlike Java arrays, collections can store any number of elements. The simplest collection is **ArrayList**, which you can treat as a resizable array. The following line of code creates an (initially empty) **ArrayList** of objects of class **Person**:

```
ArrayList<Person> friends = new ArrayList<Person>();
```

Note that the type of the collection includes the element type in angle brackets. This "tunes" the collection to work with the specific element type, so that, for example, the function **get()** of **friends** will return object of type **Person**. The **ArrayList<Person>** offers the following API (for the complete API see (Oracle, 2011)):

- **int size()** – returns the number of elements in the list
- **boolean isEmpty()** – tests if this list has no elements
- **Person get(int index)** – returns the element at the specified position in this list
- **boolean add(Person p)** – appends the specified element to the end of this list
- **Person remove(int index)** – removes the element at the specified position in this list

- `boolean contains( Person p )` – returns true if this list contains a given element
- `void clear()` – removes all of the elements from this list

The following code fragment tests if the `friends` list contains the person `victor` and, if, `victor` is not there, adds him to the list:

```
if( !friends.contains( victor ) )
  friends.add( victor );
```

All collection types support iteration over elements. The simplest construct for iteration is a "for" loop. Suppose the class `Person` has a field `income`. The loop below prints all friends with income greater than 100000 to the model log:

```
for( Person p : friends ) {
  if( p.income > 100000 )
    traceln( p );
}
```

Another popular collection type is `LinkedList`. The structure of a linked list is shown in Figure 10.12.

Linked lists are used to model stack or queue structures, i.e. sequential storages where elements are primarily added and removed from one or both ends.

Consider a model of a distributor that maintains a backlog of orders from retailers. Suppose there is a class `Order` with the field `amount`. The backlog (essentially a FIFO queue) can be modeled as:

```
LinkedList<Order> backlog = new LinkedList<Order>();
```

`LinkedList` supports functions common to all collections (like `size()` or `isEmpty()`) and also offers a specific API:

- `Order getFirst()` – returns the first element in this list
- `Order getLast()` – returns the last element in this list
- `addFirst( Order o )` – inserts the given element at the beginning of this list
- `addLast( Order o )` – appends the given element to the end of this list
- `Order removeFirst()` – removes and returns the first element from this list
- `Order removeLast()` – removes and returns the last element from this list

When a new `order` is received by the distributor it is placed at the end of the backlog:

```
backlog.addLast( order );
```

Each time the inventory gets replenished, the distributor tries to ship the orders starting from the head of the backlog. If the amount in an order is bigger than the remaining inventory, the order processing stops. The order processing can look like this:

```
while( !backlog.isEmpty() ) { //repeat the code below while the backlog is not empty
  Order order = backlog.getFirst(); //pick the first order in the backlog
  if( order.amount <= inventory ) { //if enough inventory to satisfy this order
    ship( order ); //ship
    inventory -= order.amount; //decrease available inventory
    backlog.removeFirst(); //remove the order from the backlog
  } else { //not enough inventory to ship
    break; //stop order backlog processing
  }
}
```

We recommend declaring collections in active objects and experiments graphically. The **Collection** object is located in the **General** palette. All you need to do is drag it on the canvas and choose the collection and the element types. At runtime you will be able to view the collection contents by clicking its icon.

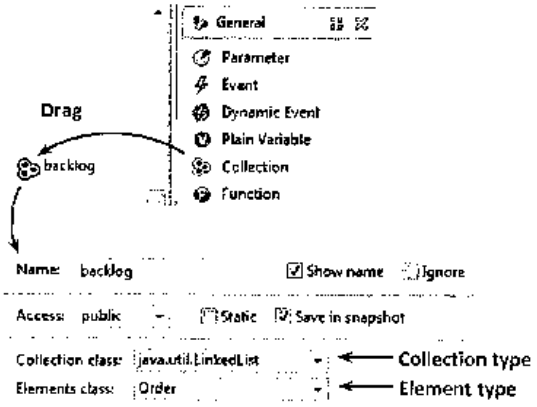


Figure 10.12 Declaring Java collection graphically

Different types of collections have different *time complexity* of operations. For example, checking if a collection of 10,000,000 objects contains a given object may take 80 milliseconds for **ArrayList**, 100 milliseconds for **LinkedList**, and less than 1 millisecond for **HashSet** and **TreeSet**. To ensure maximum efficiency of the model execution you should analyze which operations are most frequent and choose the collection type correspondingly. The following Table contains the time complexities of the most common operations for four collection types.

Operation	ArrayList	LinkedList	HashSet	TreeSet
Obtain size	Constant	Constant	Constant	Constant
Add element	Constant	Constant	Constant	Log

Remove given element	Linear	Linear	Constant	Log
Remove by index	Linear	Linear	–	–
Get element by index	Constant	Linear	–	–
Find out if contains	Linear	Linear	Constant	Log

The terms *constant*, *linear*, and *logarithmic complexity* mean the following. Linear complexity means that the worst time required to complete the operation grows linearly as the size of the collection grows. Constant means that it does not depend on the size at all, and Log means the time grows as the logarithm of the size.

You should not treat the constant complexity as the unconditionally best choice. Depending on the size, different types of collection may behave better than others. Consider Figure 10.13. It may well happen that with a relatively small number of elements the collection with linear complexity will behave better than the one with constant complexity.

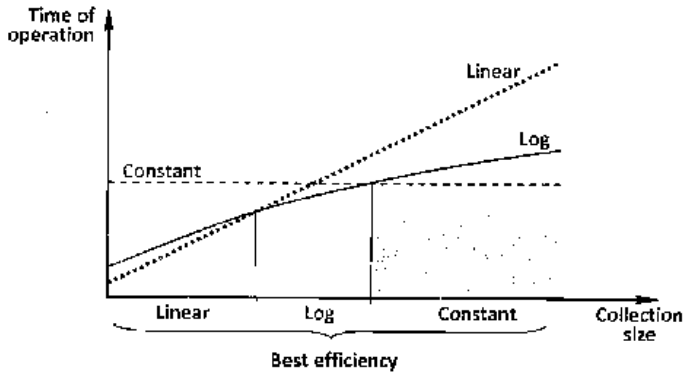


Figure 10.13 Depending on the size, different types of collections may behave better than others

And a couple of things you should keep in mind when choosing the collection type:

- **HashSet** and **TreeSet** do not support element indexes. For example, it is not possible to get an element at position 32.
- **TreeSet** is a naturally *sorted collection*: elements are stored in a certain order defined by a natural or user-provided comparator.

Replicated active objects are collections too

When you declare an embedded active object as replicated, AnyLogic creates a special type of collection to store the individual active objects. You have two options:

- **ActiveObjectArrayList** – choose this collection type if the set of active objects is more or less constant or if you need to frequently access individual objects by index.
- **ActiveObjectLinkedHashSet** – choose this option if you plan to intensively add new active objects and remove existing ones. For example, if you are modeling population of a city for a relatively long period so that people are born, die, move out of the city, and new people arrive.

The options for the type of collection appear at the bottom of the **General** page of the embedded object properties, see Figure 10.14.

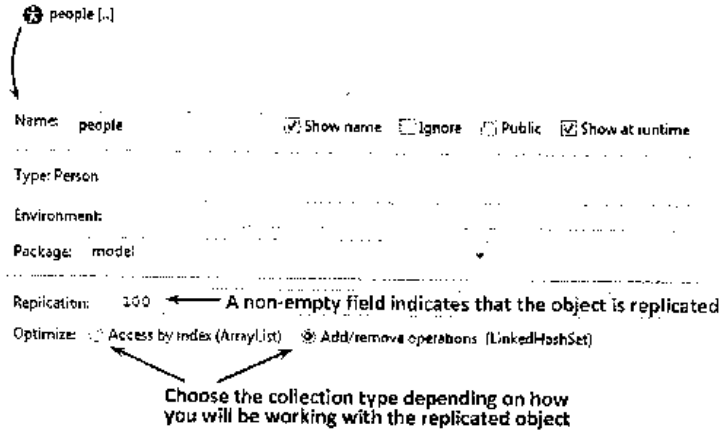


Figure 10.14 Options for collection type of a replicated object

Both collections support functions like **size(), isEmpty(), get(int index), contains(Object ao)**, and iteration. If index is not specifically needed during iteration, it is always better to use the enhanced form of "for" loop, see Section 10.8:

```
for( Person p : people ) {
  ...
}
```

10.7. Naming conventions

Now please take a moment to familiarize yourself with the naming conventions. Names you give to the model objects are important. A good naming system simplifies the development process a lot. We recommend you to keep to *Java naming conventions* (Sun Microsystems, Inc., 1999) throughout the model, not just when writing Java code.

- **The name of the object should indicate the intent of its use.**

Ideally, a casual observer should be able to understand the role of the object from its name and, on the other hand, when looking for an object serving a particular purpose, should be able to easily guess its name.

- **You can compose a name of several words. Use mixed case with the first letter of each word capitalized. Do not use underscores.**

You should never use meaningless names like "a", "b", "x23", "state1", or "event14". One-character variable names should only be used for temporary "throwaway" variables such as loop indexes. Avoid acronyms and abbreviations unless they are more common than the long form, such as: NPV, ROI, and ARPU. Remember that when using the object name in an expression or code you will not need to type it: AnyLogic code completion (see Section 10.4) will do the work for you; therefore, multi-word names are perfectly fine.

- **Keep to one naming system. Names must be uniformly structured.**

It makes sense to develop a naming system for your models and keep to it. Large organizations sometimes standardize the naming conventions across all modeling projects.

A couple of important facts: Java is a *case-sensitive* programming language: **Anylogic** and **AnyLogic** are different names that will never match. Spaces are not allowed in Java names.

In the following Table we summarize the naming recommendations for various types of model objects.

Object	Naming rules	Examples

• Java variable • Parameter of active object • Collection • Table function • Statistics • Connectivity object	First letter can be lowercase or uppercase (here we relax Java conventions), first letter of each internal word capitalized. Should be a noun. Use plurals for collections. Sometimes adding a suffix or prefix indicating the type of the object helps to understand its meaning and to avoid name conflicts. For example, **AgeDistribution** can be a custom distribution constructed from the table function **AgeDistributionTable**.	rate Income DevelopmentCost inventory AgeDistribution AgeDistributionTable friends
• Function	First letter *must* be lowercase, first letter of each internal word capitalized. Should be a verb. If the function returns a property of the object, its name should start with the word "get", or "is" for **boolean** return type. If the function changes a property of the object, it should start with "set". There are some exceptions created to make the code more compact, such as AnyLogic system functions **time()** and **date()**, or functions **size()** of Java collection.	resetStatistics getAnnualProfit goHome speedup getEstimatedROI setTarget isStateActive isVisible isEnabled
• Function argument • Local variable in the code	If possible, short, lowercase. If consists of more than one word, first letter of each internal word should be capitalized Common names for temporary integer variables are **i**, **j**, **k**, **m**, and **n**.	cost sum total baseValue i n

• Java constant	All uppercase with words separated with underscores "_"	TIME_UNIT_MONTH
Classes: • Active object class • Entity class • User's Java class • Dynamic event	First letter *must* be capitalized, first letter of each internal word capitalized as well. Should be a noun except for process model components that have a meaning of action, in which case it can be a verb.	Consumer Project UseNurse RegistrationProcess HousingSector PhoneCall Order Arrival
• Embedded object, i.e. instance of an active object class, including the library objects	First letter *must* be lowercase, first letter of each internal word capitalized. Use plurals for replicated objects.	project consumers people doTriage registrationProcess stuffAndMailBill
Dynamic variables: • Stock • Flow • Auxiliary variable	Long multi-word variable names are very common in system dynamics models. As in Java and in AnyLogic, spaces are not allowed in names; you should use other ways to separate words. We recommend using mixed case with the first letter of each word capitalized. The use of underscore "_" is not recommended, although it is allowed.	BirthRate Population DrugsUnderConsideration TimeToImplementStrategies
• Events (not dynamic) • Statechart • States • Transition	First letter can be lowercase or uppercase, first letter of each internal word capitalized.	overflow at8AMeveryDay purchaseBehavior InWorkForce discard

10.8. Statements

Actions associated with events, transitions, process flowchart objects, agents, controls, etc. in AnyLogic are written in Java code. Java code is composed of

statements. A *statement* is a unit of code, an instruction provided to a computer. Statements are executed sequentially one after another and, generally, in top-down order. The following code of three statements, for example, declares a variable x, assigns it a random number drawn from a uniform distribution, and prints the value to the model log.

```
double x;
x = uniform();
traceln( "x = " + x );
```

You may have noticed that there is a semicolon at the end of each statement. In Java this is a rule.

Each statement in Java *must* end with semicolon except for the block {...}.

We will consider these types of Java statements:

- Variable declaration, e.g.: `String s;` or `double x = getX();`
- Function call: `traceln( "Time:" + time() );`
- Assignment: `shipTo = client.address;`
- Decisions: `if( ... ) then { ... } else { ... }`, `switch( ... ) { case ... : case ...; ... }`
- Loops: `for() { ... }`, `while( ... ) { ... }`, `do { ... } while( ... )` and also `break;` and `continue;`
- Block: `{ ... }`

It is important that, regardless of its computational complexity and the required CPU time, any piece of Java code written by the user is treated by the AnyLogic simulation engine as indivisible and is executed logically instantly – the model clock is not advanced.

Variable declaration

Variable declarations are considered in Section 10.3. There are two syntax forms:

```
<type> <variable name> ;
<type> <variable name> = <initial value>;
```

Examples:

```
double x;
Person customer = null;
ArrayList<Person> colleagues = new ArrayList<Person>();
```

Please keep in mind that:

- A local variable must be declared before it is first used.
- A local variable declared within a block {...} or a function body exists only while this block is being executed.

- If the name of a variable declared in a block or function body is the same as the name of the variable declared in an enclosing (higher level) block or of the current class variable, the lower level local variable is meant by default when the name is used. We however strongly recommend avoiding duplicate names.

Figure 10.15 illustrates the code areas where local variables exist and can be used.

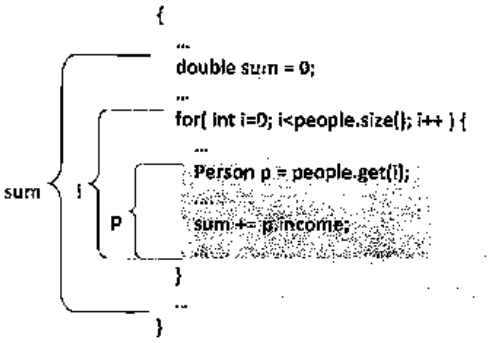


Figure 10.15 Code areas where local variables exist and can be used

Function call

Function calls are described in Section 10.4. It is worth adding that, even if a function does return a value, it still can be called as a statement if the returned value is not needed. For example, the following statement removes a person from the list of friends:

```
friends.remove( victor );
```

The function remove() returns true if the object being removed was contained in the list. If we are sure it was there (or if we do not care if it was) we can throw away the returned value.

Assignment

To assign a value to a variable in Java you write:

```
<variable name> = <expression>;
```

Remember that "=" means assignment action, whereas "==" means equality relation.

Examples:

```
distance = sqrt( dx*dx + dy*dy ); //calculate distance based on dx and dy
k = uniform_discr( 0, 10 ); //set k to a random integer between 0 and 10 inclusive
```

```
customer = null; // "forget" customer: reset customer variable to null ("nothing")
```

The shortcuts for frequently used assignments can be executed as statements as well (see Section 10.5), for example:

```
k++; //increment k by 1
b *= 2; //b = b*2
```

If-then-else

As in any language that supports imperative programming, Java has *control flow statements* that "break up the flow of execution by employing decision making, looping, and branching, enabling your program to conditionally execute particular blocks of code" (Oracle, 2013).

The "if-then-else" statement is the most basic control flow statement that executes one section of code if a condition evaluates to **true**, and another if it evaluates to **false**. The statement has two forms, short:

```
if( <condition> )
  <statement if true>
```

and full:

```
if( <condition> )
  <statement if true>
else
  <statement if false>
```

For example, this code assigns a client to a salesman only if the salesmen is currently following up with less than 10 clients.

```
if( salesman.clients.size() < 10 )
    salesman.assign( client );
```

The following code tests if there are tasks in a certain queue and, if yes, assigns the first one to a truck, otherwise sends the truck to the parking position.

```
if( tasks.isEmpty() )
  truck.setDestination( truck.parking );
else
  truck.assignTask( tasks.removeFirst() );
```

In case "then" or "else" code sections contain more than one statement, they should be enclosed in braces { ... } to become a block, which is treated as a single statement, as shown in the following code. We, however, recommend that you always use braces for "then" and "else" sections to avoid ambiguous-looking code. Braces are specifically important when there are nested "if" statements or when lines of code in the "if" neighborhood are added or deleted during editing or debugging.

```
if( friends == null ) {
```

```
    friends = new ArrayList< Person >();
    friends.add( john );
} else {
    if( !friends.contains( john ) )
        friends.add( john );
}
```

Switch

The switch statement allows you to choose between an arbitrary number of code sections to be executed depending on the value of an integer expression. The general syntax is:

```
switch( <integer expression> ) {
case <integer constant 1>:
    <statements to be executed in case 1>
    break;
case <integer constant 2>:
    <statements to be executed in case 2>
    break;
...
default:
    <statements to be executed if no cases apply>
    break;
}
```

The break statements at the end of each case section tell Java that the switch statement execution is finished. If you do not put break, Java will continue executing the next section, no matter that it is marked as a different case. The default section is executed when the integer expression does not match any of the cases. It is optional.

The integer values that correspond to different cases of the switch are usually predefined as integer constants. Imagine you are developing a model of an overhead bridge crane where the crane is an agent controlled by a set of commands. The response of the crane to the commands can be programmed in the form of a switch statement:

```
switch( command ) {
case MOVE_RIGHT:
    velocity = 10;
    break;
case MOVE_LEFT:
    velocity = -10;
    break;
case STOP:
    velocity = 0;
    break;
case RAISE:
    ...
```

```
    break;
  case LOWER:
    ...
    break;
  default:
    error( "Invalid command: " + command );
  }
```

For loop

In Java there are two forms of for loop and two forms of while loop. We will begin with the most easy to use so-called "enhanced" form of for loop:

```
for( <element type> <name> : <collection> ) {
  <statements> //the loop body executed for each element
}
```

This form is used to iterate through arrays and collections (see Section 10.6). Whenever you need to do something with each element of a collection, we recommend using this loop because it is more compact and easy to read. It is also supported by all collection types, unlike index-based iteration.

Example: in an agent based model a firm's product portfolio is modeled as a replicated object **products** (remember that replicated object is a collection). The following code goes through all products in the portfolio and kills those, where the estimated ROI is less than some allowed minimum:

```
for( Product p : products ) {
  if( p.getEstimatedROI() < minROI )
    p.kill();
}
```

Another example: the following loop counts the number of sold seats in a movie theater. The seats are modeled as Java array **seats** with elements of **boolean** type (**true** means sold):

```
boolean[] seats = new boolean[600]; //array declaration
...
int nsold = 0;
for( boolean sold : seats )
  if( sold )
    nsold++;
```

Note that if the body of the loop contains only one statement, the braces {...} can be dropped. In the code above, braces are dropped both in **for** and **if** statements.

Another, more general, form of for loop is typically used for index-based iterations. In the header of the loop you can put the initialization code. For example: the declaration of the index variable; the condition that is tested before each iteration to

determine whether the loop should continue; and the code to be executed after each iteration that can be used, say, to increment the index variable:

```
for( <initialization>; <continue condition>; <increment> ) {
  <statements>
}
```

The following loop finds all circles in a group of shapes and sets their fill color to red:

```
for( int i=0; i<group.size(); i++ ) { //index-based loop
  Object obj = group.get( i ); //get the i-th element of the group
  if( obj instanceof ShapeOval ) { //test if it is a ShapeOval – AnyLogic class for ovals
    ShapeOval ov = (ShapeOval)obj; //if it is oval, "cast" it to ShapeOval
    ov.setFillColor( red ); //set the fill color to red
  }
}
```

As long as there is no other way to iterate through the **ShapeGroup** contents than accessing the shapes by index, this form of loop is the only one applicable here.

Many Enterprise Library objects also offer index-based iterations. For example, this code goes through all entities in the queue from the end to the beginning and removes the first one that does not possess any resource units:

```
for( int i=queue.size()-1; i>=0; i— ) { //the index goes from queue.size()–1 down to 0
  Entity e = queue.get(i); //obtain the i-th entity
  if( e.resourceUnits == null || e.resourceUnits.isEmpty() ) { //test
    queue.remove( e ); //remove the entity from the queue
    break; //exit the loop
  }
}
```

Note that in this loop the index is decremented after each iteration, and correspondingly, the continue condition tests if it has reached 0. Once we have found the entity that satisfies our condition, we remove it and we do not need to continue. The **break** statement is used to exit the loop immediately. If the entity is not found, the loop will finish in its natural way when the index, after a certain iteration, becomes **-1**.

While loop

"While" loops are used to repeatedly execute some code while a certain condition evaluates to **true**. The most commonly used form of this loop is:

```
while( <continue condition> ) {
  <statements>
}
```

The code fragment below tests if a **shape** is contained in a given **group** either directly or in any of its subgroups. The function **getGroup()** of **Shape** class returns the group, which is the immediate container of the shape. For a top-level shape (a shape that is not contained in any group) it returns **null**. In this loop we start with the immediate container of the **shape** and move one level up in each iteration until we either find the **group** or reach the top level:

```
ShapeGroup container = shape.getGroup(); //start with the immediate container of shape
while( container != null ) { //if container exists
  if( container == group ) //test if it equals the group we are looking for
    return true; //if yes, the shape is contained in the group
  container = container.getGroup(); //otherwise move one level up
}
return false; //if we are here, the shape is definitely not contained in the group
```

The condition in the first form of the "while" loop is tested before each iteration: if it initially is false, nothing will be executed. There is also a second form of while loop – **do...while**:

```
do {
  <statements>
} while( <continue condition> );
```

The difference between a **do...while** loop and a **while** loop is that **do...while** evaluates its condition *after* the iteration, therefore the loop body is executed at least once. This makes sense, for example, if the condition depends on the variables prepared during the iteration.

Consider the following example. The local area map is a square 1000 by 1000 pixels. The city bounds on the map are marked with a closed polyline **citybounds**. We need to find a random point within the city bounds. As long as the form of the polyline can be arbitrary we will use the Monte Carlo method, meaning we will be generating random points in the entire area until a point happens to be inside the city. The **do...while** loop can be naturally used here:

```
//declare two variables
double x;
double y;
do {
  //pick a random point within the entire area
  x = uniform( 0, 1000 );
  y = uniform( 0, 1000 );
} while( !citybounds.contains( x, y ) ); //if the point is not inside the bounds, try again
```

Block {...} and indentation

A block is a sequence of statements enclosed in braces {...}. There can be one, several, or even no statements inside a block. Block tells Java that the sequence of

statements should be treated as one single statement, and therefore blocks are used with **if**, **for**, **while**, etc.

Java code conventions recommend placing the braces on separate lines from the enclosed statements and using *indentation* to visualize the nesting level of the block contents:

```
{
  <statement 1>
  <statement 2>
  ...
}
```

If the block is a part of a decision or loop statement, the braces can be placed on the same line with **if**, **else**, **for**, or **while**:

```
if( ... ) {
  <statements>
} else {
  <statements>
}

while( ... ) {
  <statements>
}
```

In a **switch** statement, it is recommended not to indent the lines with **case**:

```
switch( ... ) {
case ...:
  <statements>
  break;
case ...:
  <statements>
  break;
  ...
}
```

Return statement

The **return** statement is used in a function body and tells Java to exit the function. Depending on whether the function returns a value or not, the syntax of the **return** statement will be one of the following:

```
return <value>; //in a function that does return a value
return; //in a function that does not return a value
```

Consider the following function which the first order from a given client found in the collection **orders**:

```
Order getOrderFrom( Person client ) { //the function return type is Order
```

```
  if( orders == null )
    return null; //if the collection orders does not exist, return null
  for( Order o : orders ) { //for each order in the collection
    if( o.client == client ) //if the field client of the order matches the given client
      return o; //then return that order and exit the function
  }
  return null; //we are here if the order was not found -- return null
}
```

If the **return** statement is located inside one or several nested loops or **if** statements, it immediately terminates execution of all of them and exits the function. If a function does not return a value, you can skip the **return** at the very end of its body, and the function will be exited in its natural way:

```
void addFriend( Person p ) { //void means no value is returned
  if( friends.contains( p ) )
    return; //explicit return from the middle of the function
  friends.add( p );
  //otherwise the function is exited after the last statement – no return is needed
}
```

Comments

Comments in Java are used to provide additional information about the code that may not be clear from the code itself. Even if you do not expect other people to read and try to understand your code, you should write comments for yourself, so that when you come back to your model in a couple of months you will be able to easily remember how it works, fix, or update it. Comments keep your code live and maintainable; writing good comments as you write code must become your habit.

This is an example of a useless comment:

```
client = null; //set client to null – useless comment
```

It explains things obvious from the code. Instead, you can explain the meaning of the assignment:

```
client = null; //forget the client – all operations are finished
```

In Java there are two types of comments: *end of line comments* and *block comments*. The end of line comment starts with double slash // and tells Java to treat the text from there to the end of the current line as comment:

```
//create a new plant
Plant plant = add_mills();
//place it somewhere in the selected region
double[] loc = region.findLocation();
plant.setXY( loc[0], loc[1] );
//set the plant parameters
plant.set_region( region );
```

```
plant.set_company( this ); //we are the owner
```

The AnyLogic Java editor displays the comments in green color, so you can easily distinguish them from code. A block comment is delimited by /* and */. Unlike the end of line comment, the block comment can be placed in the middle of a line (even in the middle of an expression), or it can span across several lines.

```
/* for sales staff we inlclude also the commission part
 * that is based on the sales this quarter
 */
amount  = employee.baseSalary + commissionRate * employee.sales + bonus;
if( amount > 200000 )
  doAudit( employee ); /* perform audit for very high payments */
employee.pay( amount );
```

When you are developing a model it may be necessary to temporarily exclude portions of code from compilation, for example, for debugging purposes. This is naturally done by using comments. In the expression below the commission part of the salary is temporarily excluded from the expression, for example, until the commissions get modeled.

```
amount  = employee.baseSalary /* + commissionRate * employee.sales */ + bonus;
```

If you wish to exclude one or a couple of lines of code, it makes sense to put the end of line comments at the beginning of the line(s). In the code fragment below the line with the function call **ship()** will be ignored by the compiler (note that the line already contains a comment):

```
while( !backlog.isEmpty() ) { //repeat the code below while the backlog is not empty
  Order order = backlog.getFirst(); //pick the first order in the backlog
  if( order.amount <= inventory ) { //if enough inventory to satisfy this order
//  ship( order ); //ship
    inventory -= order.amount; //decrease available inventory
    backlog.removeFirst(); //remove the order from the backlog
  } else { //not enough inventory to ship
    break; //stop order backlog processing
  }
}
```

If many lines are excluded, it is better to use block comments:

```
/*
while( !backlog.isEmpty() ) { //repeat the code below while the backlog is not empty
  Order order = backlog.getFirst(); //pick the first order in the backlog
  if( order.amount <= inventory ) { //if enough inventory to satisfy this order
//    ship( order ); //ship
    inventory -= order.amount; //decrease available inventory
    backlog.removeFirst(); //remove the order from the backlog
  } else { //not enough inventory to ship
    break; //stop order backlog processing
```

```
}
}
*/
```

Be careful when using comments to exclude code, because you may make undesirable changes to the code nearby . Consider the following code fragment. The original plan was to perform an audit for every payment that is over $200,000. The model developer decided to temporarily skip the audit and commented out the line with the **doAudit()** function call. As a side effect, the next line became a part of the **if** statement and the payments will be made only to those employees who earn more than $200,000.

```
amount = employee.baseSalary + commissionRate * employee.sales + bonus;
if( amount > 200000 )
//   doAudit( employee );
employee.pay( amount );
```

Note that the accident could have been avoided if the modeler had used the block braces with the **if** statement:

```
amount = employee.baseSalary + commissionRate * employee.sales + bonus;
if( amount > 200000 ) {
//   doAudit( employee );
}
employee.pay( amount );
```

10.9. Where am I and how do I get to...?

In AnyLogic you are not writing the full code of Java classes from the beginning to the end. Instead you are entering pieces of code and expressions in numerous small edit boxes in the properties of various model elements. Therefore it is important to always understand where exactly are you writing the code (which class and method it belongs to), and how can you access other model elements from there.

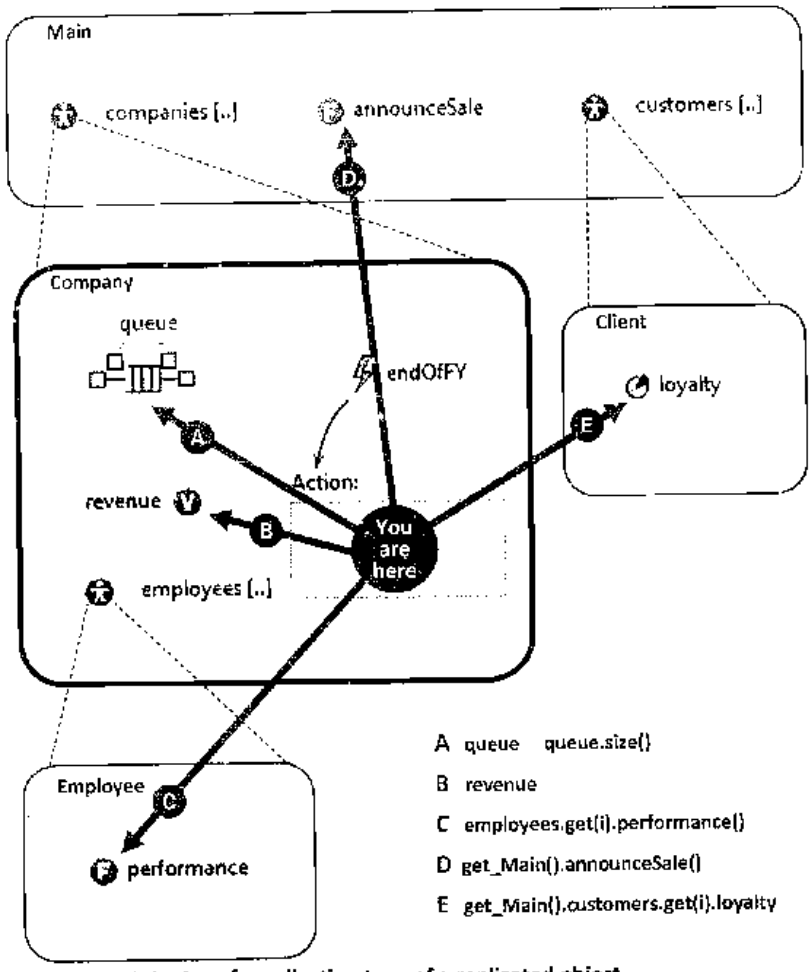


Figure 10.16 Options for collection type of a replicated object

Most of the code you write when you develop models is the code of an active object class. More precisely it is the code of one of the methods of the active object class. No matter whether you are defining an action of an event, setting a parameter of an embedded object, or writing a startup code – you can assume you are writing the code of the current active object class. Therefore the following rules apply (see Figure 10.16):

- The model elements of the same active object class are accessed simply by their names (because they are the fields of the same class). Say you are in the

Action field of the event endOfFY of class Company. To access the embedded object queue you simply write queue. To increase the variable revenue you write revenue += amount. And to restart the event endOfFY from its own action you should write endOfFY.restart(year()).

- To access an element of an embedded object, you should put dot "." after the embedded object name and then write the element name. For example, to obtain the size of the queue object you write queue.size(). If the embedded object is replicated, its name is the name of a collection of objects and you should specify exactly which one you want to access. To call the function performance() of the employee number 247 among employees of the company you should write: employees.get(246).performance().
- To access the container of the current object (the object where this object is embedded), you can call get_*<class of the container object>*(). For example, if an object of class Company is embedded into Main, you should write get_Main() to get to Main, from within Company. Consequently, to call the function announceSale() of Main you write get_Main().announceSale(). There is a function getOwner() that also returns the container object, but its return type is ActiveObject – the base class of Main, Company, etc. You will need to cast it to Main before accessing the Main-specific things: ((Main)getOwner()).announceSale(). getOwner() is useful when we do not know where exactly the current object is embedded.
- To access a "peer" object (the object that shares the container with the current one) you should first get up to the container object and then get down to another embedded object. To access a customer's loyalty from a company in Figure 10.16, we need first to get to Main and then to Client (with particular index): get_Main().customers.get(i).loyalty.

These rules naturally extend to the whole hierarchical structure of the AnyLogic model. You can access anything from anywhere. We, however, recommend developing your models in a modular way as much as possible so that the objects know minimum internals of other objects and setup, and communication is done through parameters, ports, message passing, and calls of "public" functions.

10.10. Viewing Java code generated by AnyLogic

Sometimes it may be useful to see the full Java code generated by AnyLogic to better understand the context your own code is put into. For example, the model compilation may end up with an error, which is not easy to track down by looking just at the user's code field where the error message is pointing (such errors are frequently caused by missing "{" or "}", missing or wrongly used semicolons, side effect of debug comments, etc). In AnyLogic, you can view the full Java code (.java

files) of active object classes and experiments, and easily find the place where your own portion of code is located.

To be able to view the generated Java code you need to compile the model first.

▶ **To view your code in the context of the generated Java code:**

1. Open the text edit field where you entered your code (for example, the **On enter** code of an Enterprise Library object, **Action** code of a statechart transition, dynamic field for the X coordinate of a shape, etc).
2. Press Ctrl+J. The Java file editor opens in read-only mode with your code highlighted.

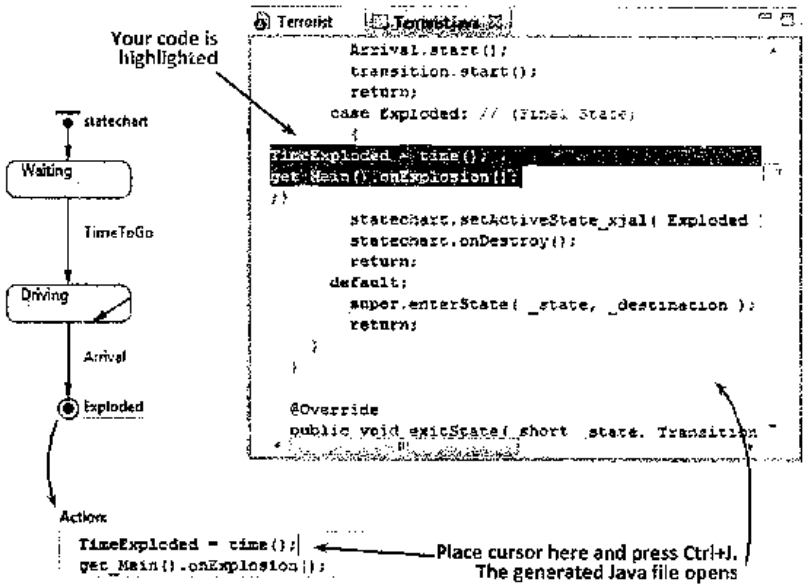


Figure 10.17 Opening the generated Java file and locating a particular portion of your code

▶ **To view the full Java file generated for an active object class or experiment:**

1. Right-click the active object class or experiment in the **Projects** tree and choose **Open with | Java editor** from the context menu.

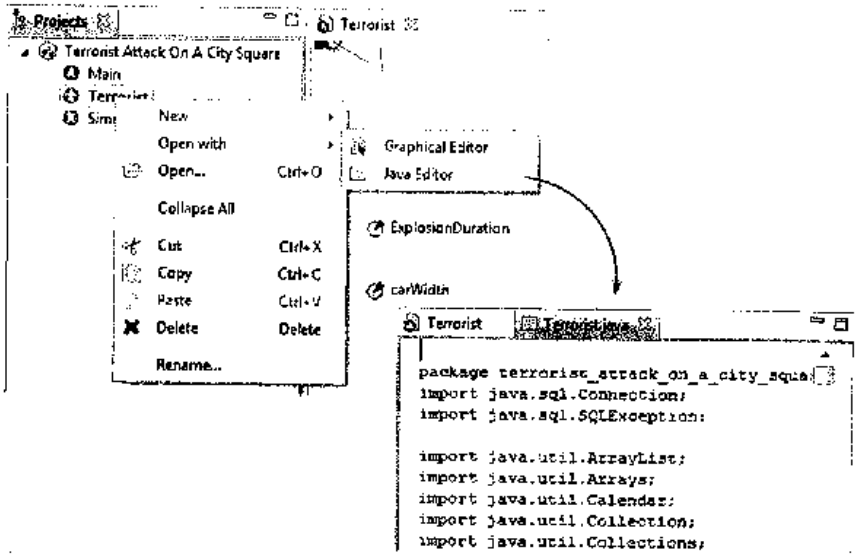


Figure 10.18 Opening the Java file generated for a particular active object class

Note that the generated Java files are for viewing purposes only. You cannot edit them in AnyLogic and should not try to edit them externally; the changes will not get back to the model.

10.11. Creating your Java classes within AnyLogic model

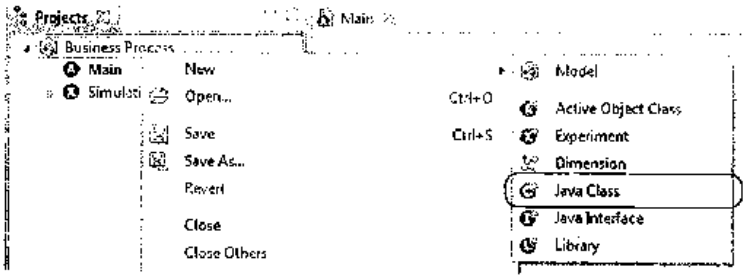
In AnyLogic you can create your own Java classes and include them in the model. This may serve various purposes. For example:

- To define entities in process models with additional fields and/or functions
- To create auxiliary data structures, such as **Location** class considered earlier
- To include classes borrowed from somewhere else in source code form and used in the model, such as problem-oriented optimization algorithms

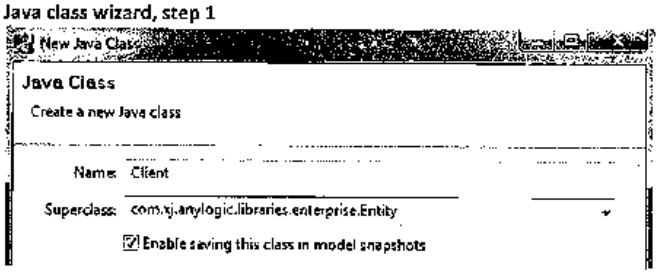
AnyLogic includes a wizard for creation of Java classes and a Java editor with syntax highlighting and code completion (the functionality is inherited from Eclipse (The Eclipse Foundation, 2013)).

As an example, we will create a Java class **Client** to be used in a business process model. **Client** will have a **boolean** field **vip** to indicate its importance, a field **complexity** of type **double** that will affect the time needed to complete the current operation requested by the client, and the field **satisfaction** where we will store the client's satisfaction level after the process is completed. As the objects of class **Client** will

flow in the business process as entities, they must have all properties of entities. Therefore we will declare the class Client as a subclass of AnyLogic class Entity.



Java class wizard, step 1



Java class wizard, step 2

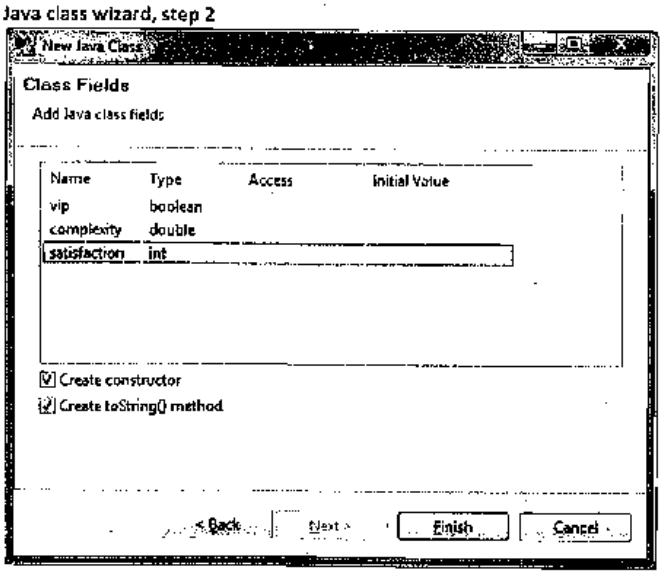


Figure 10.19 Using AnyLogic wizard to create a new Java class

▶ **To create a Java class in AnyLogic model:**

1. Right-click the model item in the **Projects** tree (the topmost one) and choose **New | Java class** from the context menu. The **New Java Class** wizard is displayed, as shown in Figure 10.19.
2. Type the name of the class (**Client**) and type or choose the superclass. The class **Entity** of AnyLogic Enterprise Library is among the available choices in the drop-down list. Press **Next**.

Until you use at least one object from the Enterprise Library in the model, the library is not "linked" to the model and its classes are not included in the drop-down list.

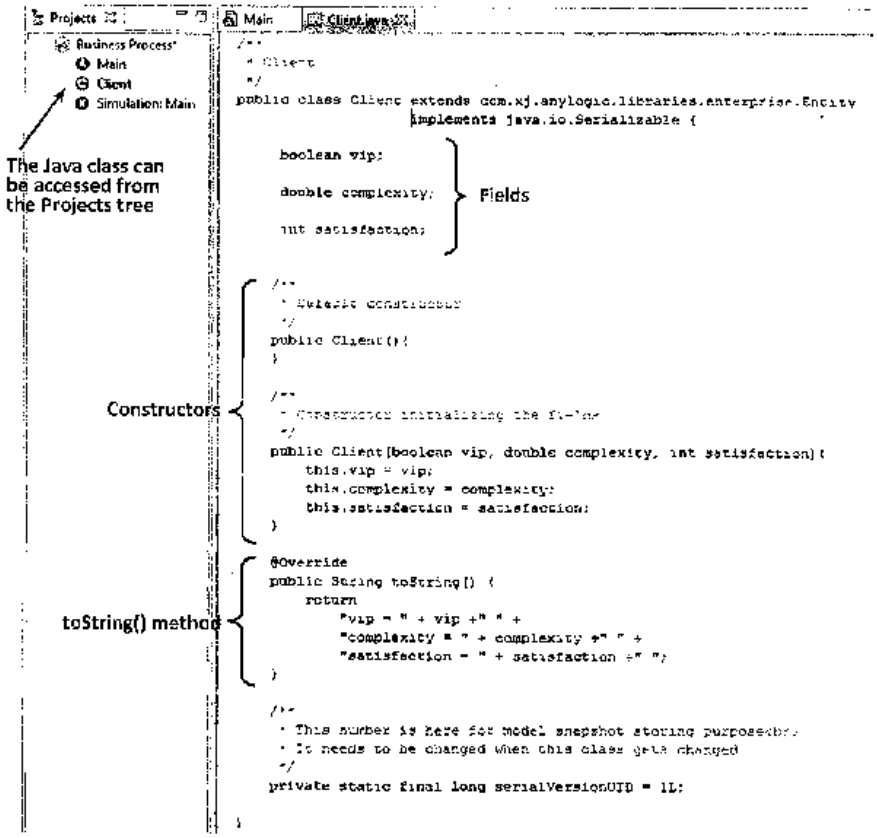


Figure 10.20 Java class created by AnyLogic wizard and opened in Java class editor

3. In the **Class Fields** page of the wizard enter the three fields that we planned to include in the **Client** class. Press **Finish**.

When the wizard completes its work, a new Java class is created in the model and opens in the AnyLogic Java class editor, a shown in Figure 10.20. The wizard generates the class declaration and the class body with fields, constructors and some straightforward implementation of the **toString()** method. You can then edit the class as you wish. The class appears in the **Projects** tree with the green icon. To open the editor, you should double-click the icon. To rename the class simply change its name in the editor and it will change in the **Projects** tree as well.

The Java class created this way belongs to the model package and therefore is in the same Java namespace as the active object and experiment classes. The name of the model package can be found and edited in the **General** page of the model properties.

Inner classes

You can also create Java classes "inside" active object classes. This may make sense when the class is simple and small and used primarily within one active object class. The entire code of the Java class is written in the **Additional class code** field on the **Advanced** page of the active object class properties. For example, in Figure 10.21, the class **BufferedCar** is defined inside the **CarSource** active object class where it is used to store cars (entities in AnyLogic Road Traffic library) with some additional information on each car in a buffer.

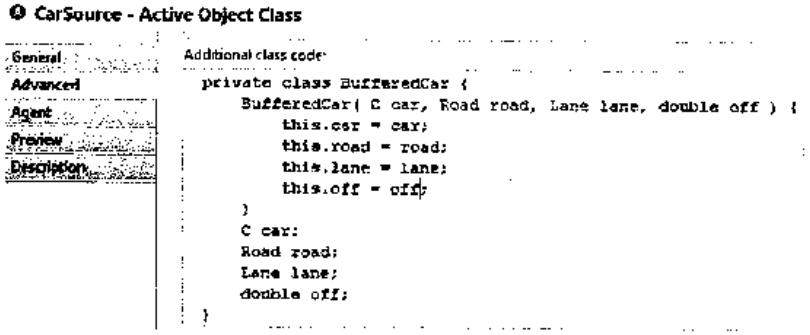


Figure 10.21 Defining an inner Java class inside an active object class

In Java, such classes defined inside other classes are called *inner classes*. Should you need to use them outside the "parent" class, you must put the name of the parent class before the name of the inner class, like this: **CarSource.BufferedCar**.

10.12. Linking external Java modules (JAR files)

The external Java code can be linked to the model in compiled form as well. Compiled Java classes are typically distributed in the packaged form in the files with .JAR extension. Numerous JAR files solving various problems can be found on the Internet, both free and commercial.

To import an external JAR file (or .class file folder) to the model:

1. Select the model (the topmost) item in the **Projects** tree and open the **Dependencies** page of its properties.
2. Press the **Add** button on the right of the list **Jar files and class folders required to build the model**.
3. In the **Add classpath entry** dialog choose the type of import, enter the file name, and choose whether to copy the file to the model folder (recommended). Press **Finish**.

Once you have imported a JAR file, you can use its classes in the model. Example "External JAR files" shows how this can be done using *JAMA package* (MathWorks & NIST, 2012)– a basic linear algebra package for Java available for free download.

Please note that when you reference the classes from the imported JAR in your model, their names must be prefixed with the package name. For example, to declare a variable of type **Matrix** defined in JAMA, you have to write **Jama.Matrix** where **Jama** is the imported package name. If you wish to use the class names without prefixes, you must write the "import" statement in the **Import** section of the **Advanced** property page of the active object class where the names are used:

```
import jama.*;
```

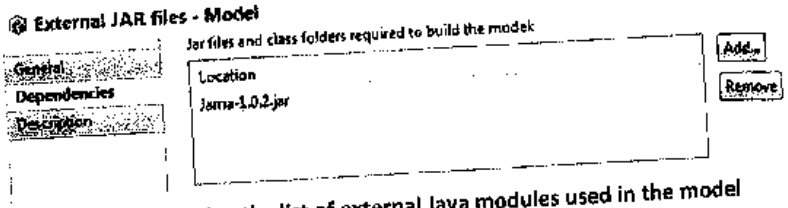


Figure 10.22 Managing the list of external Java modules used in the model

Chapter 11. Exchanging data with external world

A simulation model is virtually never a closed system. It receives various kinds of input data from outside, and outputs the simulation results. In simple cases, the model has a GUI, where the user enters parameter values in the edit boxes or changes them using sliders, and views the output charts and animation on the screen. In most industrial applications, the model reads and writes to text files, spreadsheets, or databases, or talks directly to other applications via Java.

This chapter explains how you can build data exchange interfaces to the external world. Each scenario is illustrated with an example.

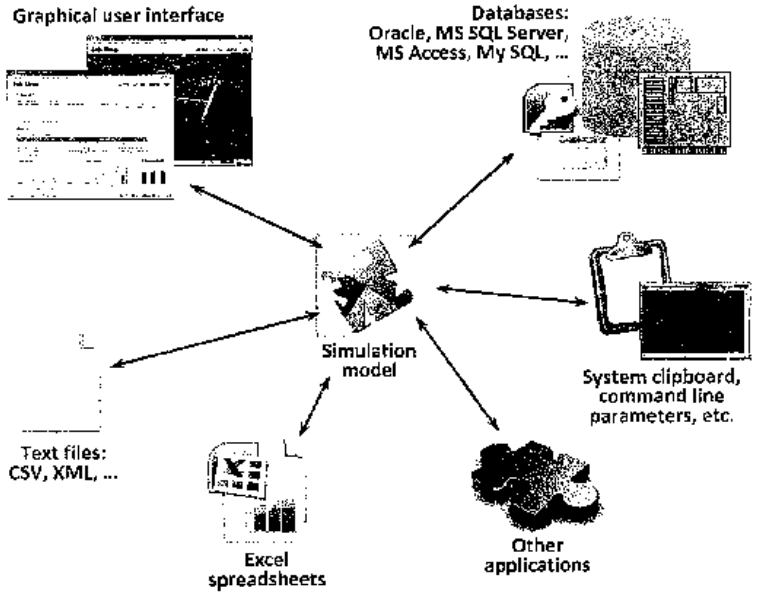


Figure 11.1 The model is virtually never a closed system

Please note that communication with the external world is fully available when you either run the models from the AnyLogic development environment, or export them as Java *applications*. If you export a model as a Java *applet* and publish it on the web, the security restrictions will prevent the applet from accessing any data (even the clipboard contents) on the end-user machine.

11.1. Text files

In simulation modeling, text files are used primarily to:

- Write unstructured output, like logs with arbitrary formatted records,
- Write output in a specific syntax format (like XML) that can be read by other applications, and
- (probably not as frequently as Excel spreadsheets) Read the model input parameters, configuration, layout, and other information.

In addition to the (always available) Java i/o API, AnyLogic offers the *text file* object that simplifies access to text files. It takes care of file opening/closing, and provides a higher-level API. The **Text file** object is located in the **Connectivity** palette (see Figure 11.2).

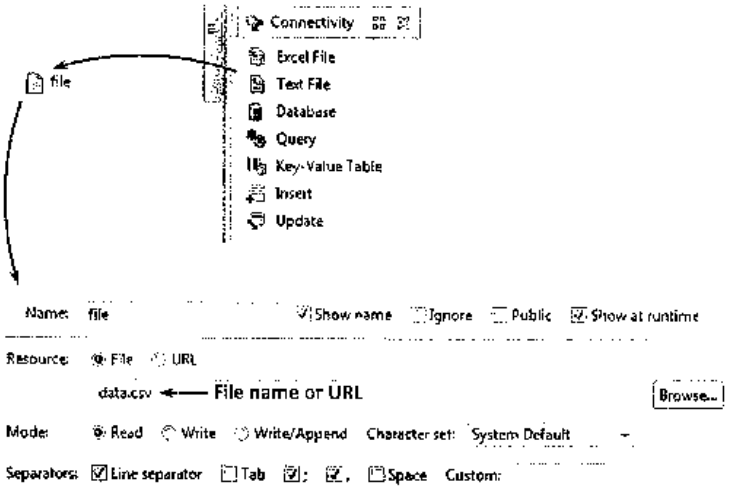


Figure 11.2 Text file object and properties

A particular file can be associated with the text file object at design time by entering its name in the text file properties, or dynamically at runtime by calling **setFile()** or **setURL()** functions. The file can be identified by the file name or by the URL. The text file object can work in **READ**, **WRITE** and **WRITE_APPEND** modes. The URL-based files are read-only.

Writing to the text files is easy. It is done by using the **print()**, **println()**, and **printf()** functions. The reading API is larger. You can read the text files byte by byte and line by line, taking care of parsing, or you can use higher level functions like **readString()**, **readDouble()**, etc. The latter way requires that you specify the *token separators* in the

file so that AnyLogic knows, for example, where the string token ends. The typical separators are listed in the text file properties, and you can provide custom ones.

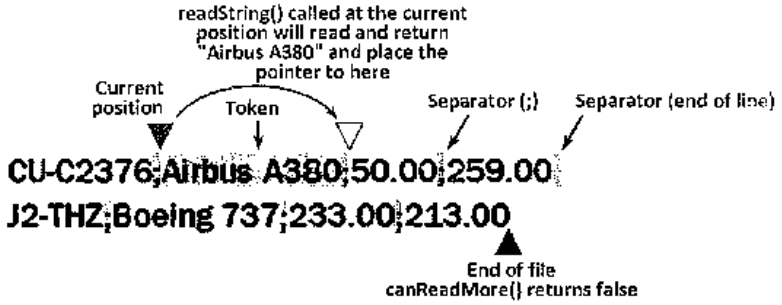

readString() called at the current position will read and return "Airbus A380" and place the pointer to here
Current position
Token
Separator (;)
Separator (end of line)
CU-C2376;Airbus A380;50.00;259.00
J2-THZ;Boeing 737;233.00;213.00
End of file
canReadMore() returns false

Figure 11.3 Tokens and separators

When you are reading a text file, there is always a pointer to the current position in the file. Each read or skip operation advances the pointer toward the end of file. The function **canReadMore()** returns **true** if there is more information to read, and **false** if the end of file is reached. This function is typically used as a **while** loop condition.

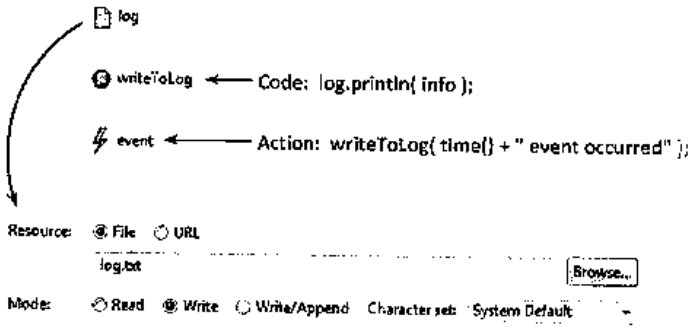
Example 11.1: Using text file as a log

This solution is an alternative to the default built-in log and may make sense if you wish to always keep the model log in a file. We will create a function **writeToLog(String info)** that will write a string to a specific text file. Upon startup, the model will check if there is a file **log.txt** in the model folder. If it exists, the model will erase its contents. If the file does not exist, the model will create a new one.

▶ **Follow these steps:**

1. Create a new model.
2. Drag the **Text file** object from the **Connectivity** palette. Change the name of the object to **log**.
3. In the text file properties, set **Mode** to **Write** and type **log.txt** in the **Resource** field.
4. Open the **General** palette and drag the **Function** object. Change the function name to **writeToLog**. Add the function argument **info** of type **String**.
5. In the function Code page, type just one line of code:
   log.println(info);
6. Click the canvas to display the properties of the **Main** active object class. In the **Startup code** field, write:
   writeToLog("Log created on " + (new Date()));
7. Drag the **Event** object from the same palette. In the event properties, change **Trigger type** to **Rate**. We will use this event to create the log contents.

8. In the Action filed of the event type:
   writeToLog(time() + " event occurred");
9. Run the model and wait until the event occurs several times.
10. Close the model.
11. Find the file log.txt in the model folder and open it in any text file viewer or editor. The file should contain a header line with the date and time the model was run, and a number of lines with times of event occurrences.



File "log.txt" created by the model

```
Log created on Sun Mar 11 13:11:17 MSK 2012
0.574595227927895 event occurred
0.5881895553808227 event occurred
1.6002410574650523 event occurred
1.7913174479492315 event occurred
2.810799260117105 event occurred
3.787106176728072 event occurred
8.367843494163509 event occurred
```

Figure 11.4 The model that uses text file as a log and sample log contents

This solution can be extended to multiple log files. In particular, if you wish to separate logs of multiple model objects, each of them can have its own log file.

Example 11.2: Reading table function from a text file

We will read the table function data from a text file, assuming each line contains a pair (argument, value). As we do not know the size of the data, we will use the text file function **canReadMore()** that returns **true** if there is some contents to read. We will use file chooser control to set up the file name to the text file object at runtime.

▶ **Follow these steps:**

1. Create a new model.
2. Drag the **Text file** object from the **Connectivity** palette. Leave its name as-is and leave the file name field blank.

3. In the properties of the text file, check all listed separators. This makes sense because we plan to read only numeric values, and can allow various separators between them.
4. Drag the **Table function** object from the **General** palette. Leave its properties as they are.
5. Drag the **File chooser** object from the **Controls** palette.
6. In the **Action** field of the file chooser, type the following code (this code is explained below):

```
//setup text file object
file.setFile( value, TextFile.READ );
//read data, use collections as data size is unknown
ArrayList<Double> arguments = new ArrayList<Double>();
ArrayList<Double> values = new ArrayList<Double>();
while( file.canReadMore() ) {
  arguments.add( file.readDouble() );
  values.add( file.readDouble() );
}
//convert collections to arrays
int N = arguments.size();
double[] args = new double[ N ];
double[] vals = new double[ N ];
for( int i=0; i<N; i++ ) {
  args[i] = arguments.get(i);
  vals[i] = values.get(i);
}
//provide the arrays to the table function
tableFunction.setArgumentsAndValues( args, vals );
```

7. Use any text file editor to create a new text file. Add several lines containing (argument, value) pairs. For example:

```
0 0
1 0.15
2 0.25
3 0.44
4 0.58
5 0.97
6 0.78
7 0.61
8 0.34
9 0
```

   Save that file with the name, say, **data.txt**.
8. Run the model.
9. Use the file chooser to choose the text file you have created (**data.txt** in this example).

10. Once you click **Open** in the file chooser, the table function should fill with the values. Check it in the popup inspect window by clicking the table function.

The code in the **Action** field of the file chooser does the following. Table functions are usually defined at design time by entering arguments and values in their properties page. Here we are changing the table function while the model is running. This is done by calling the method **setArgumentsAndValues(double[] args, double[] vals)** that accepts two arrays with element type **double**. As we are reading the table function data from a file, we do not know its size in advance, and cannot allocate the arrays before the full data is read. Therefore, we need to use variable size storage for the data that is being read, namely the Java collection **ArrayList**. When the data has been fully read, we allocate the arrays and copy the data there.

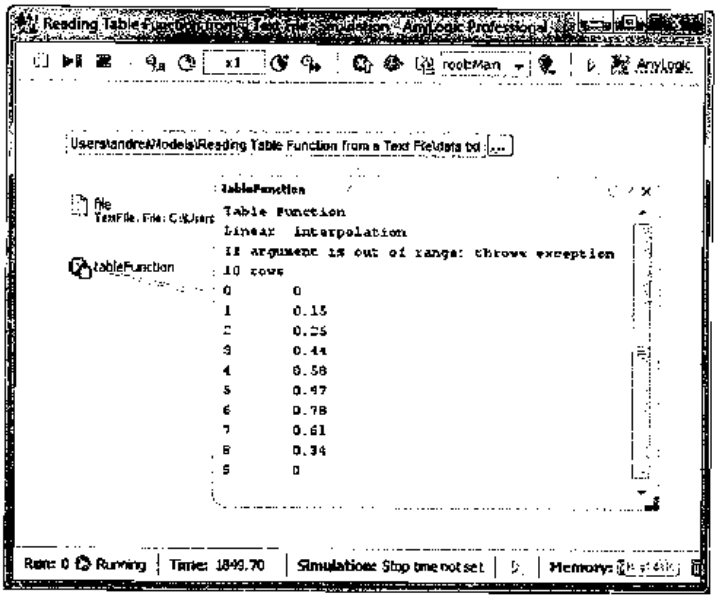


Figure 11.5 Table function read from a text file

Example 11.3: Reading agent parameters from a CSV file

Suppose you have a fleet of airplanes, and wish to read their current locations and other parameters from a file at the model startup. We will use a .csv (comma separated values) text file format.

▶ **Follow these steps:**

1. Create a new model.

2. Drag the **Agent population** object from the **General** palette to **Main**.
3. In the agent population wizard, change the class name to **Airplane**, the population name to **airplanes**, and the initial number of agents to 0.
4. Click the **airplanes[...]** icon to open the editor of the **Airplane** class.
5. In the **Airplane** class add two parameters:
   registration of type **String**, and
   type of type **String**.
6. Delete the default agent animation shape of the **Airplane** class (**person**) and replace it with the **plane** shape from the **Pictures** palette.
7. Open the **Dynamic** page of the **plane** shape properties. Enter the following expression in the **Scale X** field:
   type.equals("Airbus A380") ? 1 : (type.equals("Boeing 737") ? 0.5 : 0.25)
   Copy the same expression into the **Scale Y** field. This way, the size of the airplane picture will correspond to its type.
8. Drag the **Text** object from the **Presentation** palette, and place it nearby the airplane wing.
9. In the **Dynamic** property page of the text, enter **registration + " / " + type** in the **Text** field so each plane will display its registration number and type.
10. Open the editor of the **Main** class. Drag the **Text file** object from the **Connectivity** palette. In its properties, specify **data.csv** as the file name and select the **Line separator** , ";", and "," (CSV format can have both semicolons and commas as separators).
11. In the Startup code field of the **Main** class, write:

```
while( file.canReadMore() ) {
  String reg = file.readString();
  String type = file.readString();
  double x = file.readDouble();
  double y = file.readDouble();
  Airplane plane = add_airplanes( reg, type ); //create a new airplane agent
  plane.setXY( x, y ); //place it to the initial location
}
```

12. Create a text file with the aircraft fleet data. It makes sense to use Excel to edit the data and then save the spreadsheet in CSV format. The resulting file should contain something like this:

```
CU-C2376;Airbus A380;50.00;259.00
J2-THZ;Boeing 737;233.00;213.00
SU-XBG;Embraer ERJ-195;356.00;189.00
TI-GGH;Boeing 737;39.00;277.00
HK-6754J;Airbus A380;92.00;93.00
6Y-VBU;Embraer ERJ-195;215.00;340.00
9N-ABU;Embraer ERJ-195;151.00;66.00
```

YV-HUI;Airbus A380;337.00;77.00

Save the file in the model folder with the name **data.csv**.

13. Run the model. You should see the airplanes of various sizes located at different positions (see Figure 11.6).

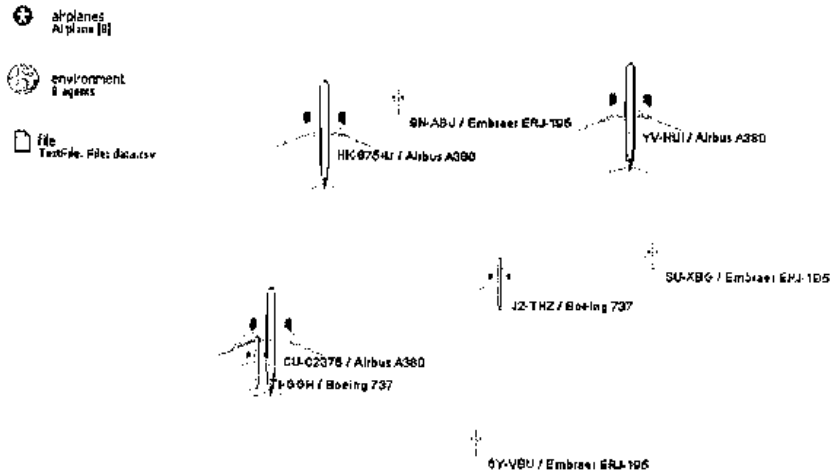


Figure 11.6 Agents representing the fleet of aircrafts created using Info from a text file

You can also use the AnyLogic **Excel file** object to work specifically with CSV files, see the Section 11.2.

There are some more examples of working with the text files in other chapters: Example 13.6: "File chooser for text files" and Example 12.11: "Read graphics from a text file".

11.2. Excel spreadsheets

Excel is probably the most common end-user interface used to input data into the simulation model and, in many cases, for displaying results as well.. Similar to text files, AnyLogic provides an **Excel file** object to enable easy access to Excel files. This object supports Excel 1997-2007 formats (both XLS and XLSX file extensions). It is also located in the **Connectivity** palette.

The **Excel file** object works as follows. Upon model startup (if the corresponding option is selected in its properties) it reads the specified Excel file and creates its internal representation inside the AnyLogic model (see Figure 11.7). The model can then access the Excel spreadsheets, read and write data, create cells, etc, using the

Excel file API. All changes are made to the internal copy of the file, and are not saved to the actual file on disk until the model terminates (again, if that option is selected), or until you explicitly call the writeFile() function.

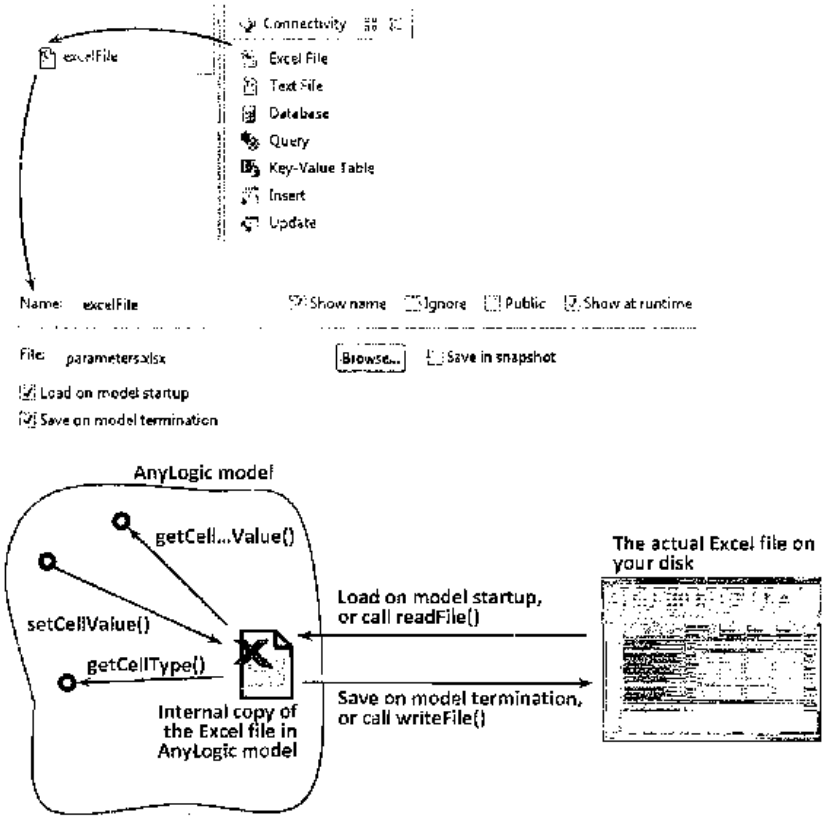


Figure 11.7 Excel file object

Excel file object API allows you to obtain the number of sheets, the number of rows and columns in a sheet, test if the cell exists, and so on. Reading data from individual cells is done by calling functions **getCellNumericValue()**, **getCellStringValue()**, **getCellType()**, etc. There are functions that read more complex structures: **readHyperArray()** and **readTableFunction()**. Similarly, there are functions writing to the Excel file: **setCellValue()**, **createCell()**, and **writeDataSet()**.

The cells can be addressed in three different ways:

- By Excel name: "**sheet1!B24**" or "**Warehouse parameters!AA96**",

- By sheet name, index of row, and index of column: **"sheet1", 2, 24,** and
- By sheet index, index of row, and index of column: **1, 2, 24.**

The last two forms are useful for iterations across Excel sheets.

Example 11.4: Reading data of various types from fixed cells in Excel

In this example, we read data of various types located in Excel cells with known and fixed coordinates into AnyLogic variables, parameters, text shapes, table functions, and controls.

	A	B	C	D	E	F	G
1							
2	**Integer value (formula =B23*B24)**	124000			**Hyper array data**	MALE	FEMALE
3					1	20	2014
4	**Real value**	15.8			2	1808	152
5					3	340	15
6	**Boolean value**	TRUE			4	150	0
7					5	18	0
8	This text variable will be set to the text shape						
9							
10	**The data on the right will be read into the table function**	0	0				
11		1	0.15				
12		2	0.25				
13		3	0.44				
14		4	0.58				
15		5	0.97				
16		6	0.78				
17		7	0.61				
18		8	0.34				
19		9	0				
20							
21	**The date that will be used to schedule event**	15.07.2020					
22							
23	The values on the right are used in the formula of B2	124					
24		1000					
25							

Figure 11.8 Excel file with various kinds of data

The Excel file that we will read, **data.xlsx**, is shown in Figure 11.8. There are data in numeric, Boolean, text, and date formats. There is also one formula in the cell B2.

The reading is done upon model startup by the following code written in the **Startup code** section of the **Main** active object class:

```
//read integer variable from B1 - need to cast from double to int
variable = (int)excelFile.getCellNumericValue( "Sheet1!B2" );
//read double(real) value into a parameter; use set_... method instead of assignment
set_parameter( excelFile.getCellNumericValue( "Sheet1", 4, 2 ) );
//set checkbox state from cell with a Boolan value
checkbox.setSelected( excelFile.getCellBooleanValue( 1, 6, 2 ) );
//set text in a text shape to the text from Excel
text.setText( excelFile.getCellStringValue( "Sheet1!A8" ) );
//read table function - we need to specify the number of rows
excelFile.readTableFunction( tableFunction, "Sheet1!B10", 10 );
//read hyper array (only one and two-dimensional arrays can be read)
```

```
excelFile.readHyperArray( hyperarray, "Sheet1!F3", false ); //first dim across columns
//schedule event to occur at time/date read from Excel
event.restartTo( excelFile.getCellDateValue( "Sheet1!B21" ) );
```

The result is shown in Figure 11.9. Reading individual cells is quite straightforward. For the table function, you have to give the number of rows. For the hyper array, you need to indicate whether the first dimension spans across rows (last parameter is true) or columns (false).

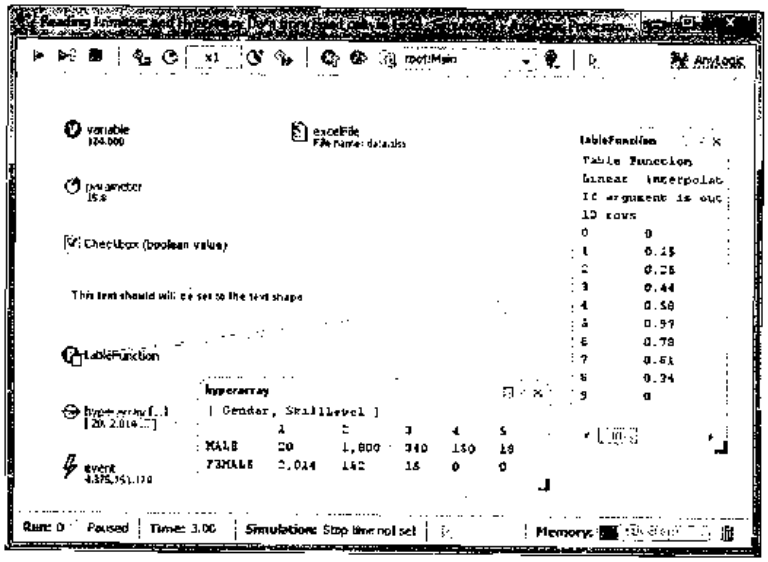


Figure 11.9 The data has been read into the model

Formula values are calculated and kept at the time of file loading. If, at runtime, you change the value of a cell that is a part of a formula, the formula will not be automatically evaluated. You need to explicitly call the function **evaluateFormulas()**.

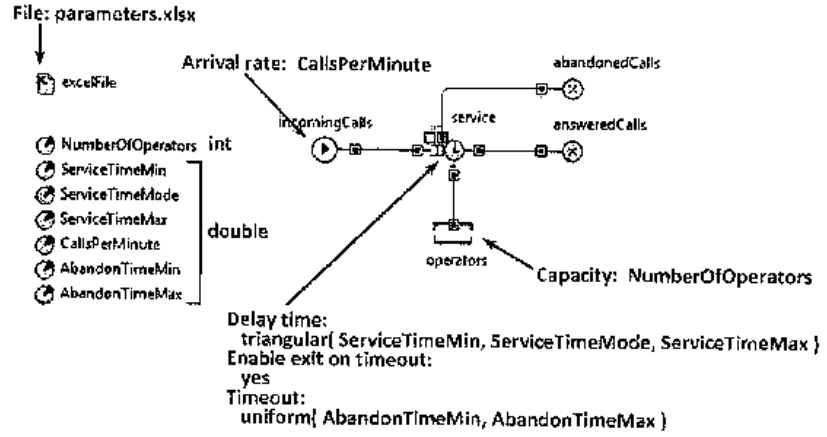
For example, the following code changes B23, evaluates all formulas, and re-reads the value of B2 into the variable:

```
//change vlaue of cell B23
excelFile.setCellValue( 250, "Sheet1!B23" );
//force formula evaluation
excelFile.evaluateFormulas();
//refresh the value of the variable
variable = (int)excelFile.getCellNumericValue( "Sheet1!B2" );
```

Example 11.5: Reading model parameters from Excel using Java reflection

Suppose there is an Excel workbook containing various model parameters. The parameter values are located in particular columns but in arbitrary rows, so *we do not know the exact cell for a parameter*. We will use parameter names located in the cells next to their values to identify and set up the parameters.

We will iterate through the spreadsheet, read the parameter name, look up the corresponding parameter in the AnyLogic model, find out its type, read the parameter value, and set it to the parameter. We will use the Java *reflection* feature to search for the parameter by name. Reflection allows us to access a class field or method *by name* at runtime.



parameters.xlsx

	A	B	C
1	Description	Name	Value
2	Number of operators	NumberOfOperators	20
3	Minimum service time, minutes	ServiceTimeMin	0.4
4	Most likely service time, minutes	ServiceTimeMode	3
5	Maximum service time, minutes	ServiceTimeMax	15
6	Calls per minute	CallsPerMinute	6
7	Minimum abandon time, minutes	AbandonTimeMin	5
8	Maximum abandon time, minutes	AbandonTimeMax	10
9			

Call center

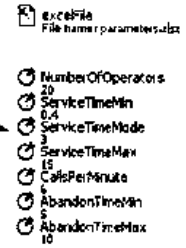


Figure 11.10 The model parameters are read from Excel file on the model startup

Consider a very simple process (discrete event) model of a call center (see Figure 11.10). Calls come in at a certain constant rate, namely 6 calls per minute. 20 operators answer the calls. The call duration is distributed triangularly between 0.4 and 15 minutes with 3 as the most likely value. If the waiting time gets too long, clients abandon calls. Time to abandon is distributed uniformly between 5 and 10

minutes. All these parameter values are not hardcoded in the model; they are read from the Excel file.

The Excel file has one sheet, "Call center", with the following structure. There is a header row with column names. The column "Description" contains a freeform description of the parameters. The column "Name" contains the parameter names spelled exactly as they are spelled in AnyLogic. This will be the key for the parameter setup. The last column, "Value", contains the parameter values of different types.

The code that reads and sets up the parameters is written in the Startup code field of the **Main** object:

```
//we will need the current (Main) class "descriptor" data to use reflection
Class c = getClass();
//find columns with parameter names and values by reading the header row
String sheet = "Call center";
int colnames = 1;
int colvalues = 1;
while( ! excelFile.getCellStringValue( sheet, 1, colnames ).equals( "Name" ) )
  colnames++;
while( ! excelFile.getCellStringValue( sheet, 1, colvalues ).equals( "Value" ) )
  colvalues++;
//read parameters starting with row 2 (just below header row)
int row = 2;
while( excelFile.cellExists( sheet, row, colnames ) ) {
  //get parameter name
  String name = excelFile.getCellStringValue( sheet, row, colnames );
  //find parameter with that name
  try {
    //get field with this name
    java.lang.reflect.Field f = c.getField( name );
    //prepare data for the the set_... method
    java.lang.reflect.Method m;
    String methodname = "set_" + name;
    //get type of the field, read value and call the "set_..." method
    Class<?> fc = f.getType();
    if( fc.equals( int.class ) ) { //integer type
      m = c.getMethod( methodname, int.class );
      m.invoke( this, (int)excelFile.getCellNumericValue( sheet, row, colvalues ) );
    } else if( fc.equals( double.class ) ) { //double type
      m = c.getMethod( methodname, double.class );
      m.invoke( this, excelFile.getCellNumericValue( sheet, row, colvalues ) );
    } else if( fc.equals( String.class ) ) { //String type - use generic method "set"
      m = c.getMethod( methodname, String.class );
      m.invoke( this, excelFile.getCellStringValue( sheet, row, colvalues ) );
    } else if( fc.equals( boolean.class ) ) { //boolean type
      m = c.getMethod( methodname, boolean.class );
```

```
        m.invoke( this, excelFile.getCellBooleanValue( sheet, row, colvalues ) );
      } else {
        error( "This parameter type cannot be read from Excel: " + fc );
      }
    } catch( Exception e ) {
      error( "Could not setup the parameter: " + name + " because of:\n" + e );
    }
    //to the next row
    row++;
  }
```

In this code, we first locate the columns with parameter names and values, and then read the parameters one by one until the last row in the sheet.

We cannot just assign the value to the parameter (like **NumberOfOperators = 20**), because the change handler will not be invoked. In particular the parameters of the embedded objects linked to that parameter will not automatically change. The only proper way of changing the parameter value is to call its **set_...** method: **set_NumberOfOperators(20);**.

Therefore, we are constructing the **set_...** method for each parameter and invoking it using reflection. This operation requires exception handling, so it is wrapped with **try...catch**. As different parameter types have different arguments in the **set_...** method, we need to treat each type individually.

Example 11.6: Displaying the model output as a chart in Excel

We will build a simple system dynamics model of the spread of contagious disease, and display its output in Excel. We will first create the Excel file with a formatted chart referencing a range of (so far empty) cells, and then fill those cells with data upon model termination. In the model, the data will be collected into a dataset, so we will use the function **writeDataset()** of the Excel file object.

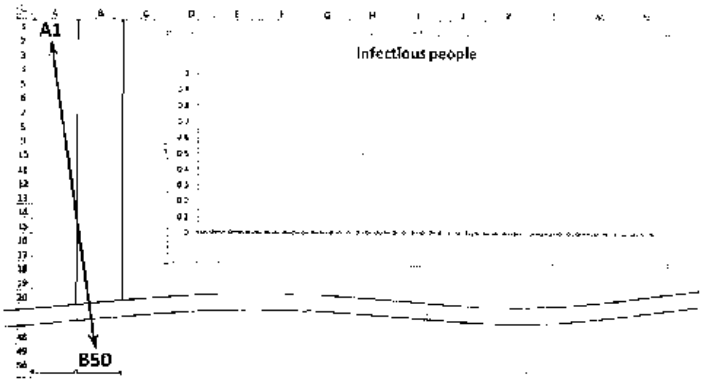


Figure 11.11 Excel sheet with a chart prepared to accept the data from the model

▶ **Follow these steps:**

1. Create an Excel file **output.xlsx** with a chart with a data set "Infectious people", referencing the cell range A1:B50 (see Figure 11.11). Close the file.
2. Create a system dynamics model with three stocks: **Susceptible**, **Infectious**, **Recovered**, and parameters as shown in Figure 11.12.
3. Add an **Excel file** object referencing the file **output.xlsx**, and make sure both checkboxes **Load on model startup** and **Save on model termination** are selected.

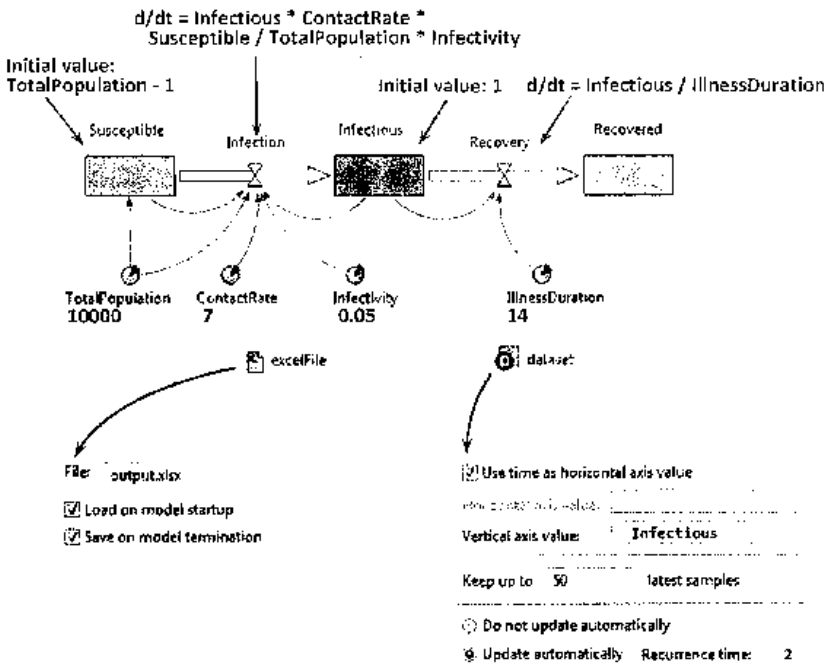


Figure 11.12 AnyLogic model set up to write the output data to the Excel file

4. Add a **Dataset** object from the **Analysis** palette. Leave time as a horizontal axis value and enter **Infectious** as a vertical axis value (we will display the number of infected people over time as the model output). Let the dataset keep 50 samples and update automatically every 2 time units.
5. In the **Destroy code** of the **Main** object write:
   excelFile.writeDataSet(dataset, 1, 1, 1);
   which means that, upon simulation termination, the model will write the dataset into the Excel file starting with the row 1 / column 1 of the first sheet.
6. In the **Model time** page of the simulation experiment properties, select **Stop at the specified time**, and set **Stop time** to **100**. In the **Presentation** page of the experiment properties, set **Virtual time**.
7. Run the model (make sure that the file **output.xlsx** is not open in Excel while the model is running). The model (working in virtual time mode) should finish almost immediately. Close the model window or press the **Stop** button to let the model write its output.
8. Open the Excel file again. The chart should now display the curve, see Figure 11.13.

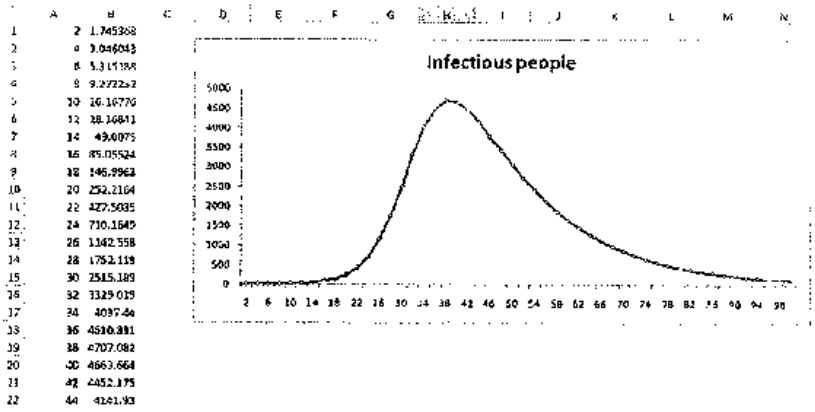


Figure 11.13 Excel sheet with the model output

If you run the model again (perhaps with different parameters), it will overwrite the previous output in the Excel file.

If you write more data (say, a dataset of size 100), the chart in Excel will not automatically adjust its cell range – you will need to do it manually.

In this example, we were dumping the whole dataset to Excel, and therefore we were using the function **writeDataset()**. Should you need to output individual data, you can use the function **setCellValue()** to write to an existing cell, and you may need to call **createCell()** if the cell does not exist.

11.3. Databases

Databases are much more powerful at data management than Excel spreadsheets. They can store very large amounts of data and support flexible and efficient data search and retrieval. Simulation models use databases primarily if:

- There is a *large amount of regularly structured* data. For small and irregularly structured data, like shown in Figure 11.13, it makes sense to use Excel.
- You want to take advantage of queries to extract information.
- This is a requirement, for example, when the database is being used by other applications (such as ERP, CRM, etc) with which the simulation model needs to communicate.

Usage of databases, however, requires some knowledge. All modern, widely-used databases (for example, Oracle, MS SQL Server, MySQL, MS Access) are *relational databases* and support SQL (Structured Query Language). You can consider a

relational database as a set of tables. A table has a number of columns, and each column has a name (unique in the table) and data type (like numeric, text, date, etc). The data is stored in table rows (also called records). Records are added and deleted as you work with the database. A table with N columns can be considered as N-ary relation between data items, hence relational database.

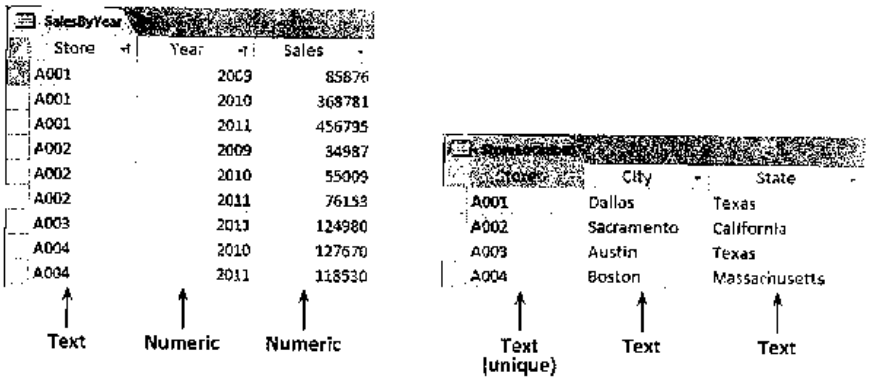


Figure 11.14 A database with two tables

Consider Figure 11.14. A database has two tables: **SalesByYear** and **StoreLocation**. The first one contains information about stores' annual sales in recent years.

Note that in this table there is one column, **Year**, and not multiple columns for 2009, 2010, etc. For each store, therefore, there are multiple records, each telling how big the store sales were in a particular year. This may look unnatural, especially for those who are used to Excel tables "growing in both dimensions". The database "style" of data structuring, however, is to keep the number of columns fixed and small, in order to enable uniform SQL access to the data.

The second table tells where the stores are located.

Note that in relational databases there are *no links* between the tables or within one table. The name "A002" in the table **SalesByYear** is just a text, and so is the name "A002" in the table **StoreLocation**. The correspondence between tables is established, if needed, by comparing the cell values.

You may have noticed that, in the first table, there is no column where the data items have to be unique (only the combination of **Store+Year** should be unique if the data is correct). In the second table, the first column **Store** should contain unique values.

Having a "key" column with unique values is *not* a requirement for a table in a relational database. In some implementations, however, you can mark a column as unique to avoid errors.

SQL queries

When you are using a database, you rarely need the full data from the original tables. Typically, you want to extract data satisfying a particular condition. SQL helps you to define such queries to the database.

It is important that once you have submitted the SQL query to the database, it is *compiled and executed at the database server*, and only the result is being transmitted to the requesting machine. This way you save a lot of network and CPU bandwidth.

Suppose you want to obtain the list of stores from our database whose sales in 2011 exceeded $100,000. This is done by the following SQL query:

```
SELECT Store, Sales
  FROM SalesByYear
    WHERE Year = 2011 AND Sales > 100000
```

The SELECT *<column name(s)>* FROM *<table name>* statement extracts data and returns it in the form of a table. We are extracting two columns: the store name and the sales amount. All necessary information is located in a single original table SalesByYear. The WHERE clause puts an additional condition on the data being extracted (it filters the query result). In our case, we are interested only in the records for 2011 in the Year column and those where the value of the Sales column is greater than 100,000. The result is shown in Figure 11.15.

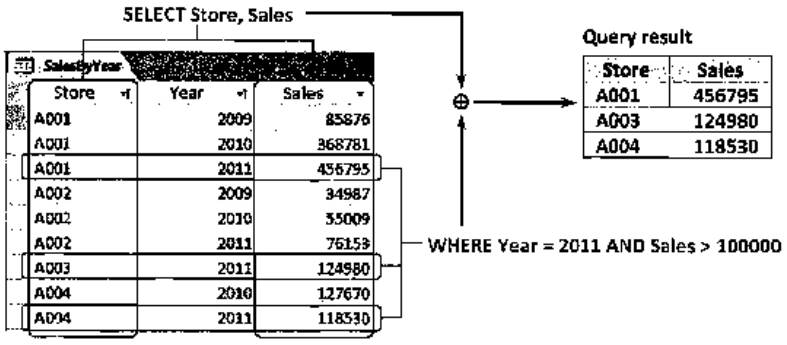


Figure 11.15 The result of a simple SQL query

Consider a slightly more complex case: we still want to single out the stores with revenue higher than $100,000 in 2011, but only those located in Texas. This time, we need to access both SalesByYear and StoreLocation tables. As the SELECT statement works with only one table, we need to *join* those tables into one by matching the store name, and then select the stores satisfying the new condition. The corresponding SQL query will look like this:

```
SELECT SalesByYear.Store, Sales, City
  FROM SalesByYear INNER JOIN StoreLocation ON SalesByYear.Store=StoreLocation.Store
    WHERE Year = 2011 AND Sales > 75000 AND State = 'Texas'
```

As the Store column is present in both SalesByYear and StoreLocation tables, it needs to be prefixed by the table name to avoid ambiguity. The result of JOIN operation (the temporary table) is shown in Figure 11.16. It is the basis for further filtering the records according to the specified condition.

As you can see, SQL is actively manipulating tables. It may create temporary intermediate tables as the query is being executed, and returns result in the table form as well.

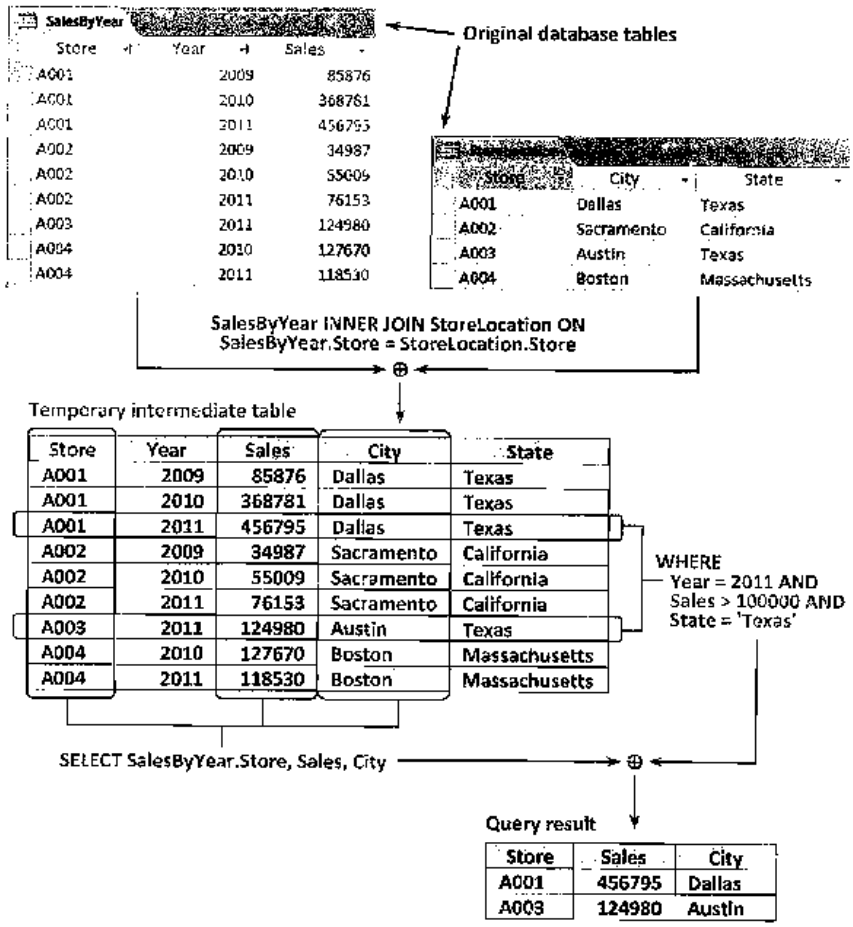


Figure 11.16 A more complex SQL query creates a temporary table

SQL can also be used to *modify* the database. Suppose you wish to update the sales of store A002 in year 2011. This is done by the **UPDATE** statement:

```
UPDATE SalesByYear
  SET Sales = 81245
    WHERE Store = 'A002' AND Year = 2011
```

In the **SET** section you can specify the list of columns where you want to change values: **SET** *<column=value, column=value...>*. The **WHERE** clause selects a row, or multiple rows, in which those column values should be changed. Be careful: if you omit the **WHERE** clause, the new values will be set to *all rows in the table*!

Another useful SQL statement is **INSERT**. It adds a new record (row) into a table. For example, to add new sales results of store A003 in the year 2012, you can write:

```
INSERT INTO SalesByYear
  VALUES ( 'A003', 2012, 211088 )
```

Note that, although in Figure 11.17 the new record is shown added as the last row of the table, in relational databases the notion of *order of records in the table does not exist*. You should treat the records in a table as an *unordered set*. You can, however, ask to deliver the query results ordered.

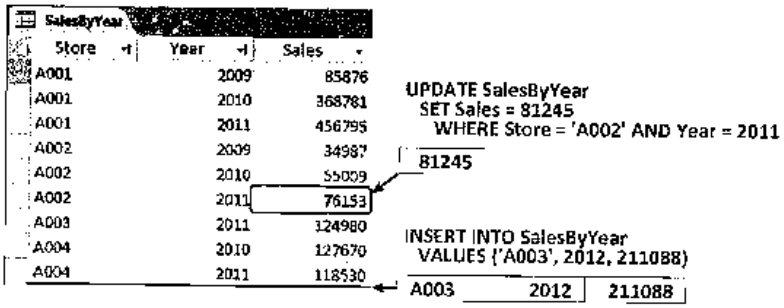


Figure 11.17 SQL statements used to modify the database

There are a lot of books and online references on SQL; for example, (Refsnes Data, 2013). Also, a particular database may support specific SQL dialects.

AnyLogic database connectivity objects

AnyLogic offers five objects in the **Connectivity** palette that simplify access to databases (see Figure 11.18). The **Database** object establishes a connection to the actual database. It sets up the JDBC driver and stores the login and password (if needed). This object has rich Java API, and is sufficient to do any database-related work in the model. The other four objects are designed to perform simple operations, namely:

- **Query** extracts data from the database and returns it as the Java object **ResultSet**. Optionally, it can add an agent or add a Java object into a collection, one per each record in the result set. Agent parameters or Java object fields will be set to the record field values.
- **Key-Value table** builds a two-column table in the AnyLogic model from a database table. One column is considered a key, another a value. You are then able to retrieve values simply by calling its method **get(** *<key>* **)**.
- **Insert** inserts a record into a database table.

- **Update** updates a field in a table. The record is chosen based on the value of another column (a "key" column should exist).

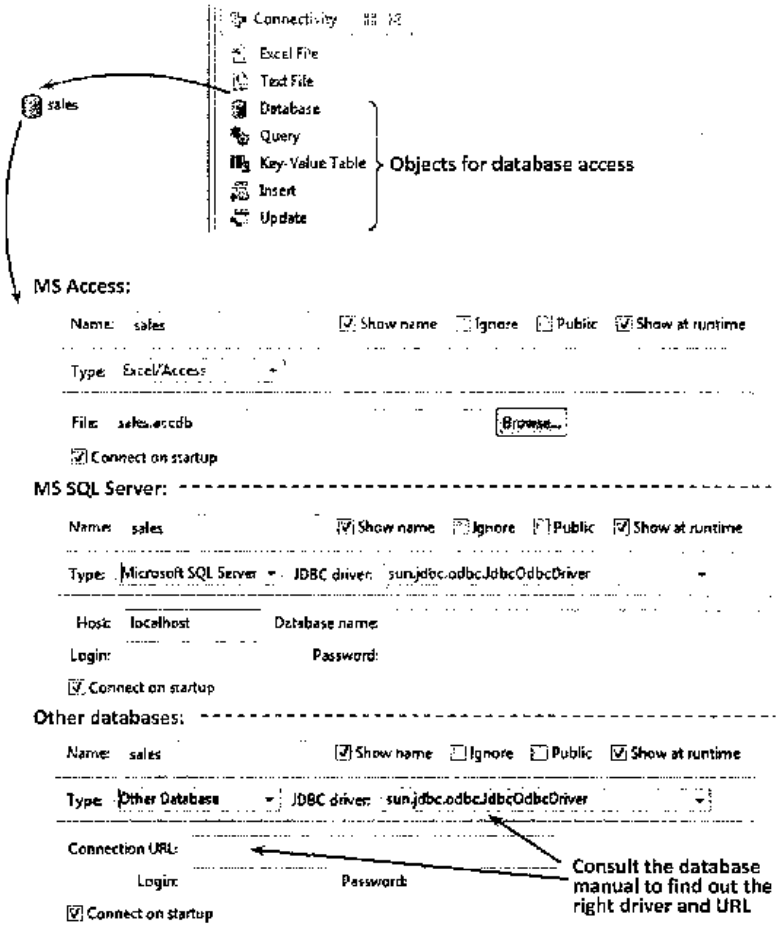


Figure 11.18 Properties of the Database object

For non-primitive operations, however, usage of the database-related AnyLogic API (in particular, the ResultSet class) would be needed, and some knowledge of SQL would be helpful.

Example 11.7: Loading data from a database and using ResultSet

We will use the database we considered above. We will establish a connection to the database using the AnyLogic **Database** object, execute some queries, and iterate through the query results.

▶ Follow these steps:

1. Create an MS Access database with two tables as shown in Figure 11.19. Save it in the file "Stores and Sales.accdb".
2. Create a new model.
3. Open the **Connectivity** palette and drag the **Database** object. In its properties, type **StoresAndSales** in the **Name** field, set **Type** to **Excel/Access**, and choose **Stores and Sales.accdb** in the **File** field. Leave the checkbox **Connect on startup** selected.
4. Drag the **Button** object from the **Controls** palette. Set the button **Label** to "Select whole table".
5. In the button **Action**, type the following code:

```
ResultSet rs = StoresAndSales.getResultSet( "SELECT * FROM StoreLocation" );
while( rs.next() ) {
  traceln( rs.getString( "Store" )+": "+rs.getString( "City" )+", "+rs.getString( "State" ) );
}
```

6. Run the model. Once the model is running, the info string under the database name should read "Connected" (see Figure 11.19).
7. Press the button and open the **Console** window of the AnyLogic model development environment. The console should contain the list of stores with their locations.

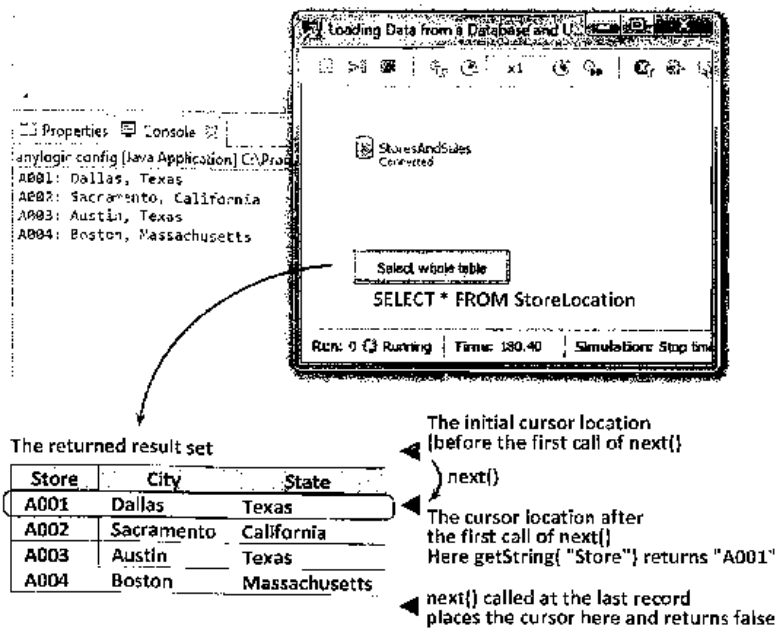


Figure 11.19 Iterating through the result set

In the action code of the button, we called the method **getResultSet()** of the database object, which executes a given SQL query. The query has the form **SELECT * FROM** *<table name>*, where "*" means that *all columns* of the table are selected. As no additional conditions were specified (the **WHERE** clause is omitted), the result will contain the whole original table.

The result of a query returned as an object of the Java class **ResultSet**. This class has a number of methods to iterate across records (rows) and retrieve individual field values. The result set has a notion of *cursor*. In the just returned "untouched" result set, the cursor is located *before the first record*, and you need to call **next()** or **first()** to place it at the first record. Typically, you iterate through the result set using **while** loop with **next()** as a condition: **next()** moves the cursor one record forward and returns **true** while the cursor is pointing at an existing record. After the last record, the cursor will move to "after the last record" position, and **next()** will return **false**.

When the cursor is pointing to a record, you can retrieve the record field values by using a variety of **get...()** methods. There is a method for each data type: like **getDouble()**, **getDate()**, **getString()**, etc. You can use the column name or column index as an argument. In the above example, **getString("Store")** is equivalent to **getString(1)**.

Notice that, in databases, all numbering starts at 1, not at 0 like in Java.

For our next query, we will define in the same model using the AnyLogic Query object. The Query object is visual and, having defined it once, you can call it from different places. Also, as you will see in the next example, it can automatically create agent populations and collections using this object.

▶ **Follow these steps:**

1. Drag the Query object from the Connectivity palette and call it HighRevenuesIn2011InTexas.
2. In the Database field of the query properties, write StoresAndSales.
3. Select SQL form of the query, and type the following code underneath:

```
SELECT SalesByYear.Store, Sales, City
  FROM SalesByYear INNER JOIN StoreLocation ON SalesByYear.Store =
     StoreLocation.Store
   WHERE Year = 2011 AND Sales > 100000 AND State = 'Texas'
```

4. Drag the Button object from the Controls palette. Set the button Label to "Select highest TX revenues in 2011".
5. In the button Action, type the following code:

```
traceln( "\nStores with revenue > 100,000 in 2011 in Texas\n" );
ResultSet rs = HighRevenuesIn2011InTexas.execute();
while( rs.next() ) {
  traceln( rs.getString( "Store" ) + ": " + rs.getDouble( "Sales" ) +
     " in " + rs.getString( "City" ) );
}
```

6. Run the model, press the newly created button, and open the Console window. The console should contain the following text:

```
Stores with revenue > 100,000 in 2011 in Texas

A001: 456795.0 in Dallas
A003: 124980.0 in Austin
```

Example 11.8: Creating agent populations parameterized from a database

The Query object can also be used to create populations of agents (see Section 3.2) and fill collections of Java objects (see Section 10.6) without writing any code. The idea is that an agent (or a Java object) is created for each record in the result set, and gets parameterized by the fields of record.

TruckLocations

TruckID	X	Y
R546_T	563	240
S691VG	333	115
H129BE	444	142
W472TB	195	232
Z148HT	286	372
O184TY	102	176
I546UG	478	326
D840II	195	196
S822FK	518	375
G975PL	229	194
Y589FE	198	157
P137NS	140	223

TruckTypes

TruckID	Type
A642VK	SCANIA R500
B497AP	VW Crafter
B698BJ	VW Crafter
C498PB	SCANIA R500
C634NW	MB Vario
C984JA	SCANIA R500
D840JI	VW Crafter
F783VU	SCANIA R500
G600OV	VW Crafter
G975PL	SCANIA R500
H129BE	SCANIA R500
H408QR	VW Crafter
I461JS	MB Vario

Figure 11.20 The Truck Fleet database

We will use the MS Access database **Truck Fleet** with two tables: **TruckLocation** and **TruckType** (see Figure 11.20). There are just three possible truck types in the fleet: SCANIA R500, Volkswagen Crafter, and Mercedes Benz Vario. For each truck in the database, we will create an agent of class **Truck**, set its animation according to type, and place it into the given location on the map.

▶ **Follow these steps:**

1. (Assuming you have the database ready) Create a new model. Drag the **USA Map** picture from the **Pictures** palette, and enlarge it so that if fits nicely into the 800x600 default window (see Figure 11.23).
2. Open the **General** palette and drag the **Agent Population** object. In the wizard, type the agent class name **Truck** and population name **fleet**. Set the initial population size to 0 (no truck will be created by default). Choose the environment option **Do not use**. Place the agent animation (so far this is a blue person) at the coordinate origin (see Figure 11.21).
3. Open the editor of **Truck** class. Add four parameters:
   id of type **String**,
   type of class **String**,
   x of type **double**, and
   y of type **double**.

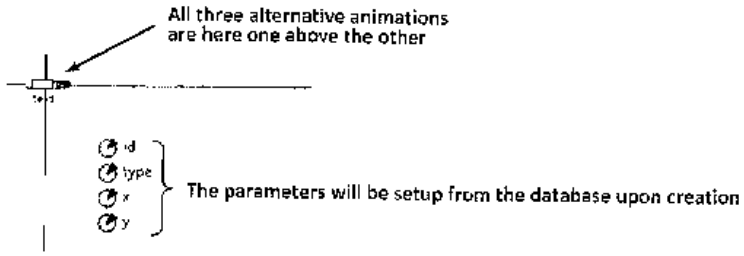


Figure 11.21 The Truck class

4. Delete the default animation of class **Truck** (the blue person). Open the **Pictures** palette and drag three pictures: **Lorry**, **Lorry 2**, and **Truck**. Reduce them to about 20% of the original size, and place them all at the coordinate origin (see Figure 11.21).
5. Open the **Dynamic** page of the **lorry** picture properties. In the **Visible** field, type **type.equals("MB Vario")**. Similarly, for the **lorry2**, type **type.equals("VW Crafter")** and for the **truck**, type **type.equals("SCANIA R500")**. This will select one of the three pictures at runtime, depending on the vehicle type.
6. Drag the **Text** object from the **Presentation** palette and place it nearby the truck animation pictures.
7. In the **Dynamic** property page of the text enter **id** in the **Text** field, so each truck displays its id.
8. Open the **Agent** page of the **Truck** class properties. Uncheck the checkbox **Environment defines initial location**, and set the initial coordinates **X** and **Y** to **x** and **y**.
9. Open the editor of the **Main** class again.
10. Open the **Connectivity** palette, and drag the **Database** object. Give it the name **TruckFleetDB**. Choose the MS Access file **Truck Fleet.accdb**.
11. Drag the **Query** object from the same palette. In its properties, set the **Database** to **TruckFleetDB**, and choose **SQL** query option.

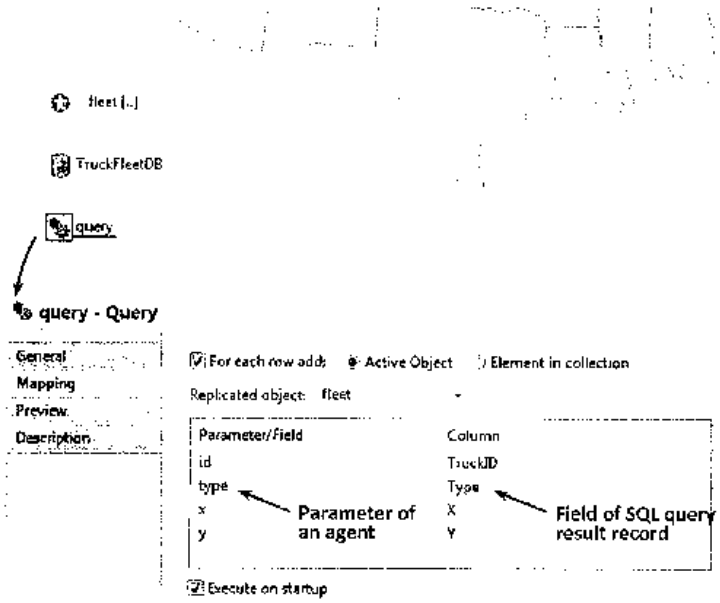


Figure 11.22 Mapping of the record fields of the SQL query result to the agent parameters

12. Type the following SQL query:

```
SELECT TruckTypes.TruckID, Type, X, Y
  FROM TruckTypes INNER JOIN TruckLocations
    ON TruckTypes.TruckID = TruckLocations.TruckID
```

13. Switch to the **Mapping** page of the query properties. Select **For each row add Active Object**, and choose **fleet** in the **Replicated object** list.
14. Specify the following parameter to field mapping:
    id : TruckID
    type : Type
    x : X
    y : Y
15. Check the **Execute on startup** checkbox.
16. Run the model. You should see the trucks of different types created all over the map.

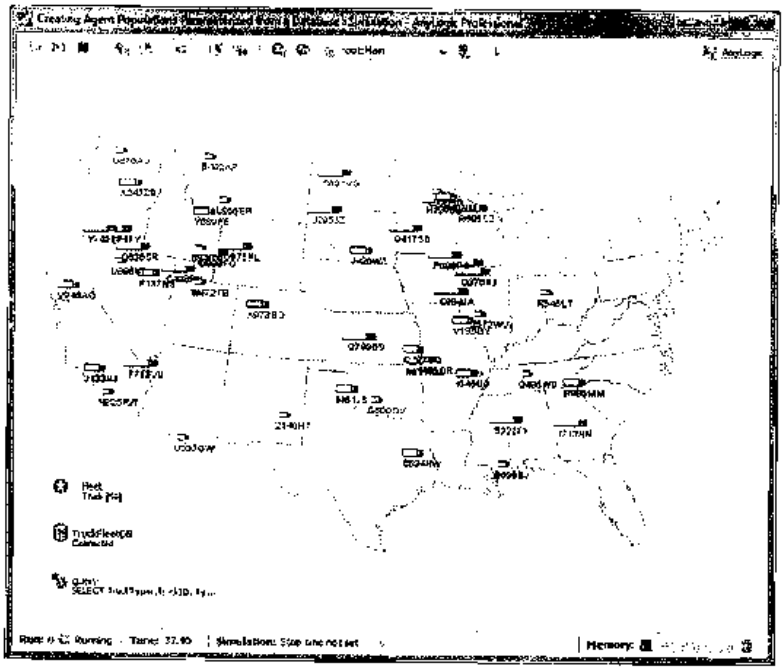


Figure 11.23 The agents of class Truck created and parameterized from the database Truck Fleet

Alternatively, instead of filling in the population of agents, you can use the Query object to fill a collection of Java objects (see Section 10.6). In that case, the mapping will have the fields of the object on the left and fields of the SQL result record on the right.

Example 11.9: Dumping simulation output into a database table

We will create a simple transaction processing model, run it for a certain time, and write all statistics to the database. The database will initially contain two tables: one with the main metrics, and another with the details of each transaction processing (see Figure 11.24).

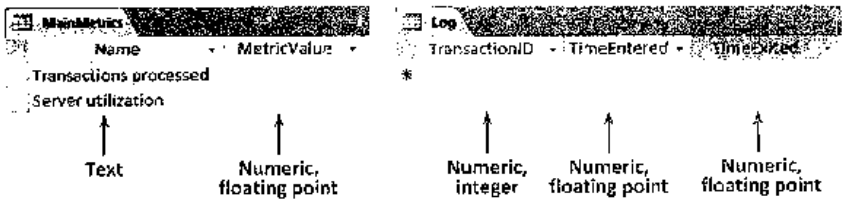


Figure 11.24 Database with two tables prepared to accept the simulation output

Note that in the table **MainMetrics**, the column with values has the name **MetricValue** and not **Value**. This is because the word 'Value' is reserved in SQL. Also, keep in mind that SQL is *case-insensitive*.

▶ **Follow these steps:**

1. Prepare an MS Access database with two tables, as shown in Figure 11.24. Save it in the file **Output Data.accdb**.
2. Create a new model. Put together a simple service model as shown in Figure 11.25. Set the server **Capacity** to **2** and check the **Enable statistics** checkbox in the server properties.

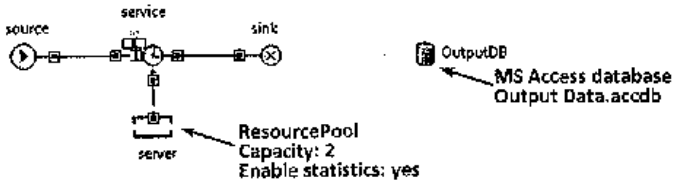


Figure 11.25 A simple service system

3. Open the **Model time** page of the **Simulation** experiment properties. Set **Stop at the specified time** and set **Stop time** to **1000**. In the **Presentation** page of the experiment properties, set **Execution mode** to **Virtual time (as fast as possible)**.
4. Open the **Connectivity** palette, and drag the **Database** object. Give it the name **OutputDB**. Choose the MS Access file **Output Data.accdb**.
5. In the **General** page of the **Main** class properties, type the following code in the **Destroy code** field:

```
OutputDB.modify( "UPDATE MainMetrics SET MetricValue = "
   + sink.count() //sink object counts entities that finish service
   + " WHERE Name = 'Transactions processed'" );
OutputDB.modify( "UPDATE MainMetrics SET MetricValue = "
   + server.statsUtilization.mean() //this is average utilization over the entire run
   + " WHERE Name = 'Server utilization'" );
```

6. Run the model. The simulation finishes almost immediately, since we are in virtual time mode. The **Destroy code**, however, is not executed until you press the Stop button or close the model window. Do so and open the database. The simulation results are stored in the MainMetrics table.

The method **modify()** of the **Database** object that we call in the **Destroy code** executes the SQL query that modifies the database, like **UPDATE** or **INSERT**. Alternatively, in this particular case we could use the **Update** object from the **Connectivity** palette. Now we will write to the database the detailed log of transaction processing.

7. Open the **General** page of the **Main** class properties, and type this code in the **Startup code** field:

```
OutputDB.modify( "DELETE FROM Log" );
```

8. Select the **source** object and type this code in its **On exit** field:

```
OutputDB.modify( "INSERT INTO Log VALUES ( "
   + entity.hashCode() + ", " + time() + ", 0 )" ); // the last 0 is just a placeholder
```

9. Select the **sink** object and type in its **On enter** field:

```
OutputDB.modify( "UPDATE Log SET TimeExited = "
   + time() + " WHERE TransactionID = " + entity.hashCode() );
```

10. Run the model, wait until it finishes, close the model window, and open the **Log** table of the database. The table now contains the full history of transaction processing (see Figure 11.26).

MainMetrics

Name	MetricValue
Transactions processed	981.00
Server utilization	0.49

Log

TransactionID	TimeEntered	TimeExited
32717744	999.474676790926	0
21168328	999.859072968192	0
610720	0.673795537670124	1.82775076642865
17030800	0.710998695755733	1.87171896222661
9110923	2.5009133413051	3.44254439587114
32644176	3.40074210338862	4.31563779618145
20054554	3.64586683274226	4.73939272611965
19559385	5.18181846420197	6.21131307515659
22447700	6.05748867068081	7.23900569757802

These two transactions were still in the system by the stop time of the simulation (1000)

Figure 11.26 The simulation output written into the database

The **DELETE** statement in the **Startup code** clears the **Log** table before the new simulation runs. The **INSERT** statement in the **source** object adds a new record to the

log table each time a new transaction enters the system. At that time, we do not know the time the transaction will exit the system, so we just set 0 as the last argument. The **UPDATE** statement in the **sink** object looks up the existing record of the transaction and updates its exit time.

The method **hashCode()** of any Java object returns its *pseudo-unique* id: there is always a probability that, in any set of objects, two or more will have the same hash code. It is OK for this demo example, but for serious models you should use a different id, like a serial number.

Example 11.10: Using prepared statement when writing to databases

You may have noticed that simulation now takes significantly more time. This is because it executes a *new* SQL statement on virtually each simulation step. Although we actually have only two requests to the database during the simulation run (to add a record for a transaction entering the system and then to update the record when the transaction exits), the database treats each request as new, because it is submitted in a string form and has to be parsed and compiled. Databases, however, support technology that allows the execution of frequent operations more efficiently. You can ask the database to *precompile statements* at the model startup, and then just execute the *prepared statements* with given parameters.

This example is a modified version of Example 11.9: "Dumping simulation output into a database table". The **INSERT** and **UPDATE** statements executed for each transaction are precompiled. The exact list of changes is below:

1. Two new variables declared at the level of the **Main** class: **insertRecord** and **updateRecord**, both of class **java.sql.PreparedStatement**. You do not need to provide initial values.
2. The **Startup code**:

```
//clear the Log table
OutputDB.modify( "DELETE FROM Log" );
try { //exception handling is necessary
  Connection con = OutputDB.getConnection();
  //turn off auto commit
  con.setAutoCommit( false );
  //prepare statements for writing into the Log table
  insertRecord = con.prepareStatement( "INSERT INTO Log VALUES ( ?, ?, 0 )" );
  updateExitTime = con.prepareStatement(
      "UPDATE Log SET TimeExited = ? WHERE TransactionID = ?" );
} catch( SQLException ex ) {
  error( ex.toString() );
}
```

3. The **Destroy** code:

```
try {
  Connection con = OutputDB.getConnection();
  //update the metrics table
  OutputDB.modify( "UPDATE MainMetrics SET MetricValue = "
      + sink.count() + " WHERE Name = 'Transactions processed'" );
  OutputDB.modify( "UPDATE MainMetrics SET MetricValue = " +
      server.statsUtilization.mean() + " WHERE Name = 'Server utilization'" );
  //commit all previous transactions
  con.commit();
  //turn on auto commit
  con.setAutoCommit( true );
} catch( SQLException ex ) {
  error( ex.toString() );
}
```

4. The **On exit** code of the **source** object:

```
try {
  //set parameters to the prepared INSERT statement
  insertRecord.setInt( 1, entity.hashCode() );
  insertRecord.setDouble( 2, time() );
  //execute it
  insertRecord.executeUpdate();
} catch( SQLException ex ) {
  error( ex.toString() );
}
```

5. The **On enter** code of the **sink** object:

```
try {
  //set parameters to the prepared UPDATE statement
  updateExitTime.setDouble( 1, time() );
  updateExitTime.setInt( 2, entity.hashCode() );
  //execute it
  updateExitTime.executeUpdate();
} catch( SQLException ex ) {
  error( ex.toString() );
}
```

Another improvement in this version is that *auto commit* is turned off at the beginning of the simulation. This means that all modifications made to the database are not considered as final until you explicitly commit them (which we do in the **Destroy code**). If the model does not finish successfully, and its output should not be placed into the database, you can ask the database to *roll back* and undo all modifications made by the model.

11.4. Working with the clipboard

As any application, a simulation model can copy and paste data to/from the system clipboard. An object in the clipboard can allow to access itself in one or multiple formats: as text, as image, as binary data, as application-specific data, etc. For example, if you copy to the clipboard a portion of an Excel spreadsheet, you can then paste it to Windows Notepad as text, to MS Word as a table, and to another Excel spreadsheet as cells.

AnyLogic supports these clipboard-related operations:

- You can copy and paste text in text controls (see Chapter 13) such as **Edit box**, **Combo box**, etc., and access the contents of those controls from the model.
- You can copy a chart data as tab-separated values.
- You can copy text by calling the function **copyToClipboard()**.
- In addition, you can access the clipboard using Java API, in particular the package **java.awt.datatransfer**.

Please note that, due to security restrictions, Java applets are not able to share the system clipboard with other applications.

In the small example, we will show how to copy and paste textual information.

Example 11.11: Working with clipboard

We will create a polyline that will be dynamically changing its shape, and a button that will copy the current polyline point coordinates to the clipboard. Another button will check if the clipboard has some contents and if the contents have a textual form. If so, the contents will be loaded into the model and displayed on the screen.

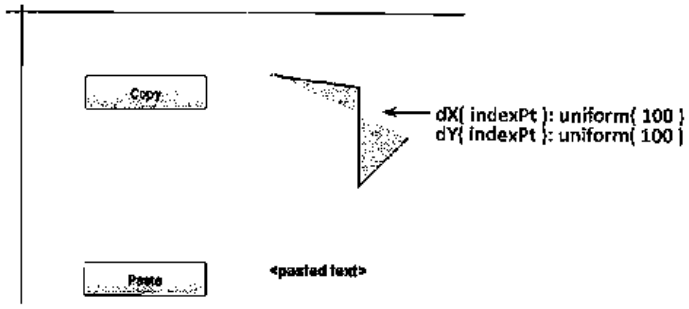


Figure 11.27 The layout of the model elements in the clipboard access model

▶ **Follow these steps:**

1. Create a new model.
2. Open the **Advanced** page of the **Main** class properties. Type this line in the Imports field:
   ```
   import java.awt.datatransfer.*;
   ```
   All clipboard-related Java classes are contained in that package, and we need to import it to enable easy access to them.
3. Open the **Presentation** palette, and drag the **Polyline** object on the canvas. Open the **Dynamic** page of the polyline properties, and set both **dX** and **dY** to **uniform(100)**. This way the polyline will randomly change its shape on each animation frame displayed.
4. Drag a button from the **Controls** palette near the polyline (see Figure 11.27). Change the button label to **Copy**, and define the action of the button:
   ```
   //put together a string with polyline coordinates
   String s = "Polyline points:\n";
   for( int i=0; i<polyline.getNPoints(); i++ )
     s += (int)polyline.getPointDx( i ) + ", " + (int)polyline.getPointDy( i ) + "\n";
   //copy the string to the clipboard
   copyToClipboard( s );
   ```
5. Drag a **Text** object from the **Presentation** palette, and place it below the polyline. Change the name of the text object to **pastedText**.
6. Create another button, label it **Paste**, and define its action:
   ```
   //clear the text
   pastedText.setText( "" );
   //get the clipboard and its contents
   Clipboard clipboard = java.awt.Toolkit.getDefaultToolkit().getSystemClipboard();
   Transferable contents = clipboard.getContents(null);
   if( contents == null ) {
     pastedText.setText( "The clipboard is empty" );
   } else if( !contents.isDataFlavorSupported( DataFlavor.stringFlavor ) ) {
     pastedText.setText( "The clipboard does not contain data in text format" );
   } else { //there is contents and it has a textual representation
     try {
       pastedText.setText( (String)contents.getTransferData( DataFlavor.stringFlavor ) );
     } catch( Exception ex ) {
       pastedText.setText( ex.toString() );
     }
   }
   ```
7. Run the model. Play with the **Copy** and **Paste** buttons. Try to copy and paste data to/from other applications.

11.5. Standard output, the model log, and command line arguments

Being a Java application, the AnyLogic model has a textual *standard output stream*, also called the *model log*. You can write there by calling these functions:

- **traceln(String text)** – outputs the text followed by the carriage return to the log,
- **trace(String text)** – outputs the text to the log but does not add a carriage return at the end, and
- **traceln()** – simply starts a new line.

The original Java function that does the same thing is **System.out.println()**. If the model is run from the AnyLogic model development environment, the log can be viewed in the Console window (see Figure 11.28).

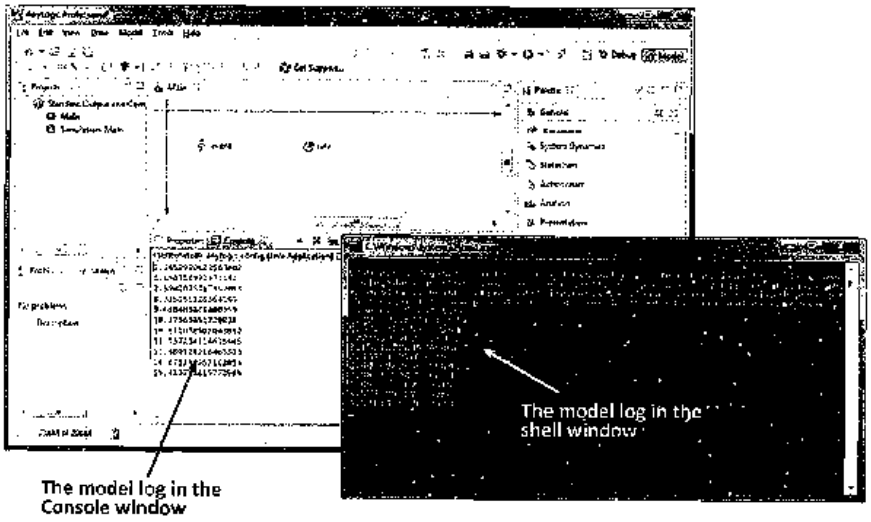


Figure 11.28 The model log (the standard output)

If the model has been exported and is run from command line, the log appears in the shell window. Alternatively, it can be streamed to a file. To do that, you need to run the .bat file generated by AnyLogic like this:

<model name>.bat > output file name

Also, when running the model this way, you can pass the *command line arguments* to the model. The arguments are placed directly after the .bat file name (and before the output stream definition, if any):

<model name>.bat arg1 arg2 arg3 ...

To access the command line arguments from inside the model, you should use the following code:

```
String[] args = getExperiment().getCommandLineArguments();
for( int i=0; i<args.length; i++ ) {
    // process parameter i...
}
```

As you can see, the command line arguments are available as an array of strings.

Chapter 12. Presentation and animation: working with shapes, groups, colors

AnyLogic offers a rich set of graphics tools to design the visual 2D and 3D front-ends of your models. The tools include various shapes (rectangles, polylines, text shapes, 2D images, etc), controls (buttons, sliders, text boxes, etc), 3D-specific elements (cameras, lights, 3D figures, etc), and also data visualization objects (charts and plots). This chapter explains how to draw 2D shapes and tells about basic animation principles.

AnyLogic is a dynamic simulation tool, and of course the picture that you draw in the AnyLogic graphical editor can be animated. Any property of any shape (such as size, position, or color) can be varied during the model runtime to reflect the dynamics of the model. Shapes can also be dynamically replicated, hidden, shown, or even created and deleted.

Graphics can do more than just decorate the model or visualize its dynamics. Graphics may also be used to define the physical or logical configuration of the model. The shapes created by the user are accessible to the model objects so that the latter can behave according to the graphical configuration. This is used in the Network Based Modeling part of the Enterprise Library where the paths and nodes drawn by the user are used to create and parameterize the network where the entities and resources are moving.

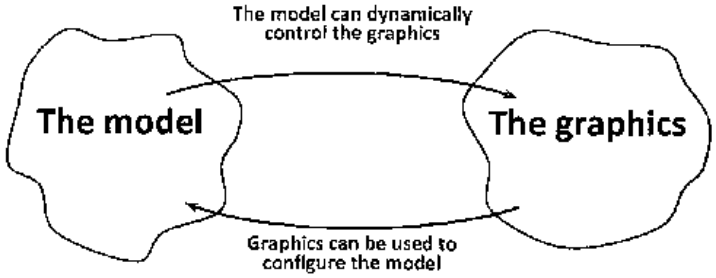


Figure 12.1 Relationship of the model and the graphics

You can consider the model objects (process flowcharts, statecharts, events, stock and flow diagrams, etc) and the graphical objects (shapes, images, groups, etc) as living in the same space and being able to access and control each other.

12.1. Drawing and editing shapes

Tools for creation of basic shapes are located in the **Presentation** palette and the simplest way to create a shape is to drag its icon from the palette and drop it on the canvas.

▶ **To create a new shape**

1. Open the **Presentation** palette.
2. Drag the shape from the palette and drop it on the canvas.

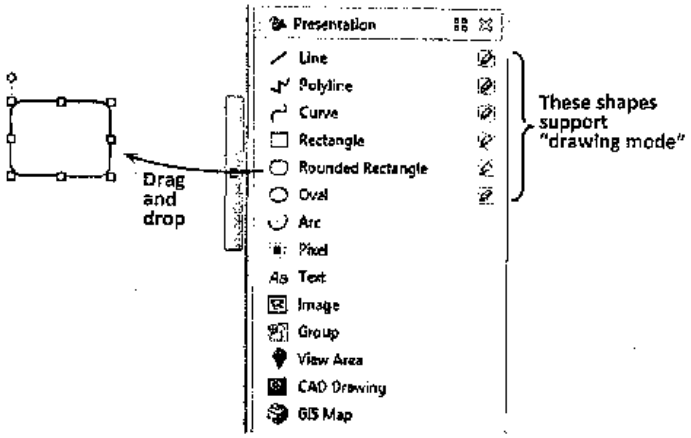


Figure 12.2 The Presentation palette

The new shape created this way has a default size and you can then adjust it as needed using handles. Alternately, you can create a shape by entering the *drawing mode*. The shapes that support drawing mode have a special icon with a pencil on the right hand side in the palette.

▶ **To create a new shape using the drawing mode (except for polyline and curve)**

1. Double click the shape icon the palette, or double click the pencil icon on the right hand side of the shape icon.
2. Click in the canvas where the upper left (or first) point of the shape should be placed.
3. Hold the mouse button; drag the cursor to the bottom right (or second) point of the shape.
4. Release the mouse button.

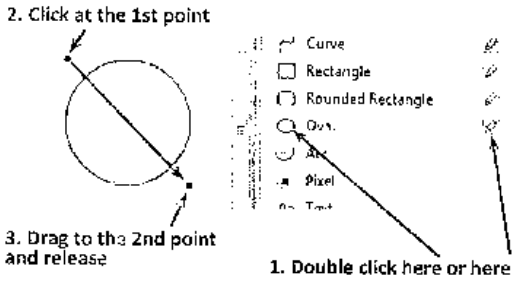


Figure 12.3 Using Drawing mode to draw shapes

Polylines and curves

Polylines and curves, are drawn and edited in a slightly different way.

▶ **To draw a polyline or a curve:**

1. Double click the polyline or curve icon in the palette.
2. Click in the canvas at the position of the first point of the polyline.
3. Continue clicking at all but the last point positions.
4. Double click at the last point position – this exits the drawing mode.

To create a closed polyline or curve you should *not* create its first and last points at the same position. Instead, you should check the **Closed polyline (Closed curve)** checkbox in the **General** page of its properties. The last edge is then drawn automatically.

Polylines and curves can be edited after creation; you can move, add and delete individual points. To do it you need to enter the *edit points* mode.

▶ **To edit individual points of a polyline or a curve:**

1. Select the polyline or curve.
2. Double click it (or choose **Edit points** from the context menu) to enter the "edit points" mode. The way the polyline (curve) is highlighted should change so that you can see the points.
3. To add a new point double click the edge where the point has to be added.
4. To delete a point double click it.
5. To move a point drag it to the new position.

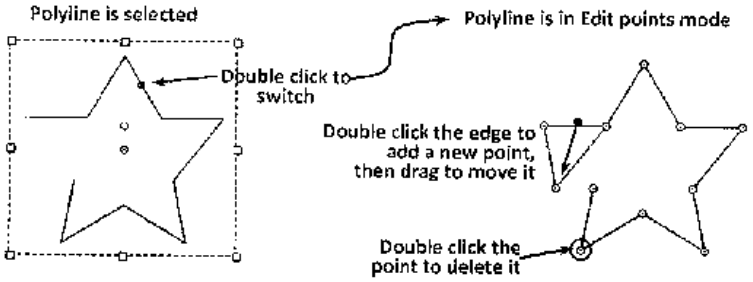

Polyline is selected
Polyline is in Edit points mode
Double click to switch
Double click the edge to add a new point, then drag to move it
Double click the point to delete it

Figure 12.4 Editing points of a polyline

The points of a polyline can also be edited in the table in the **Points** page of the polyline properties.

When you rotate a polyline or a curve in the graphical editor, the editor just recalculates the point coordinates, so there is no **Rotation** field in the design time properties of these shapes. The **Dynamic** properties page however contains a field for runtime rotation.

Arcs

Arcs have a *start angle* and an *extension angle*. While the size and position of an arc can be edited graphically by using handles, the angles are edited in the **Advanced** page of the arc properties. The angles are counted from 12 o'clock clockwise.

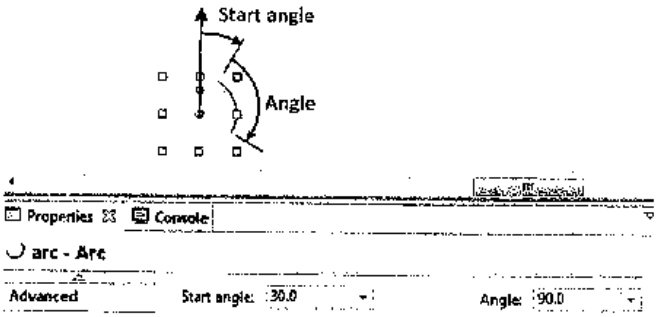


Figure 12.5 Editing arc angles

Text

Most properties of the text shape (except for position and rotation) are edited in the **General** page of its **Properties** view.

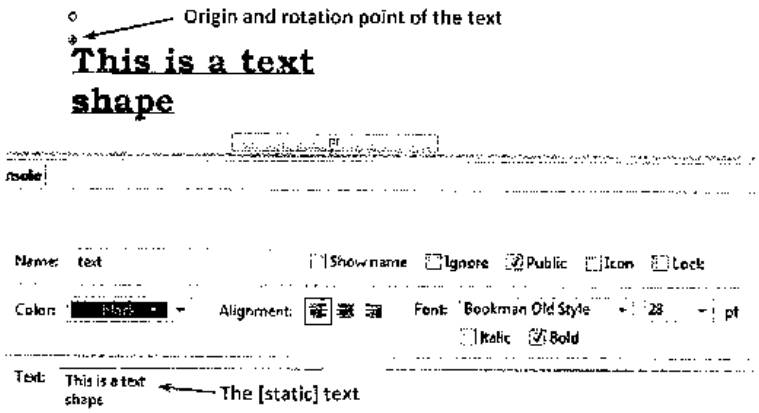


Figure 12.6 Text shape

The text will extend downwards from its origin. Depending on the alignment, it will be left justified to the right of the origin (as shown in Figure 12.6), right justified to the the left of the origin, or centered to the origin. The line spacing is chosen automatically.

The text that is entered in the **Text** field of the properties will show in design time in the graphical editor and, if not overridden by the dynamic **Text** field, also at runtime. However, if any value is provided in the dynamic **Text** field, the static text will not be used at all at runtime.

Images

When you drag the **Image** object from the palette to the canvas, you actually create a placeholder for your images. The placeholder in general may contain several images, and you can switch between them at runtime (in design time however you will see only one). This may be useful when two or more different images correspond to different (and alternative) states or modes of an object.

▶ **To add an image:**

1. Drag the image icon from the **Presentation** palette to the canvas. A placeholder for the image is created.
2. In the **General** page of the image click **Add image** button in the first (and only) empty item in the images list. The **Open File** dialog pops up.
3. Choose the image file and press **Open**. The image appears in the images list and on the canvas.
4. If needed, adjust the size of the image, or check **Original size** checkbox in the image properties to preserve its original size.

5. To add more images (that you intend to switch at runtime with the first one), click **Add Image** button on the last (and empty) item in the list and repeat previous steps.
6. To change the order of images use the control on the right above the images list.

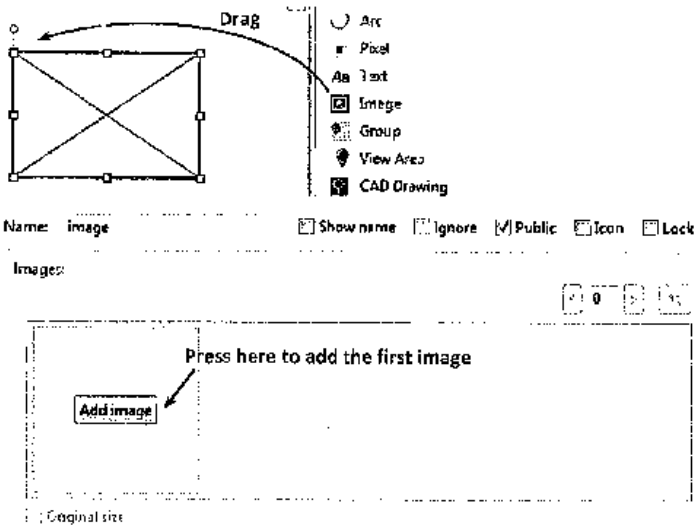


Figure 12.7 An empty image (placeholder) and its properties

The original size will be set for the first image only. The other images with different sizes will be resized to fit that size.

The image files are copied to the model folder, so that you do not have to remember where they are.

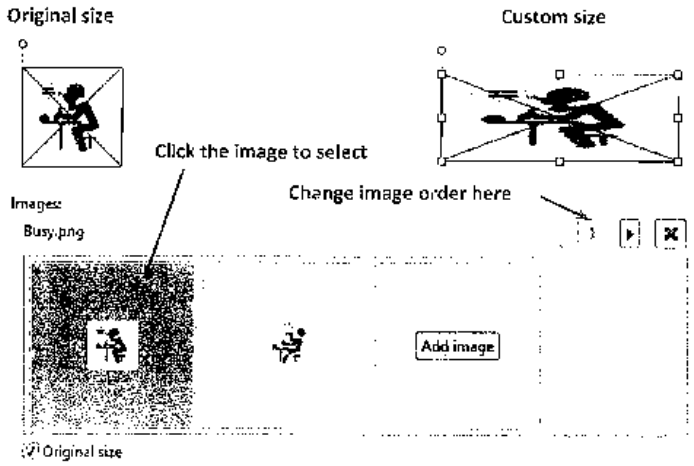


Figure 12.8 Image object with two images added

Z-Order

Although the 2D picture is flat, shapes are drawn in a certain order and shown one above the other. This order is called the *Z-order* because there is a virtual Z axis perpendicular to the surface of the picture and extending towards the viewer.

▶ **To change the Z-order of a shape:**

1. Select the shape(s).
2. Choose **Order** from the context menu
3. Choose the appropriate action:
   Bring to front places the shape on top of all other shapes
   Bring forward moves the shape one step up (swaps it with the shape directly in front)
   Send backward moves the shape one step down
   Send backward places the shape below all other shapes

Sometimes it is more convenient to use the toolbar buttons instead of the context menu.

If the shape is a member of a group, the ordering applies only within the group. The group appears as a single item in the global shape order.

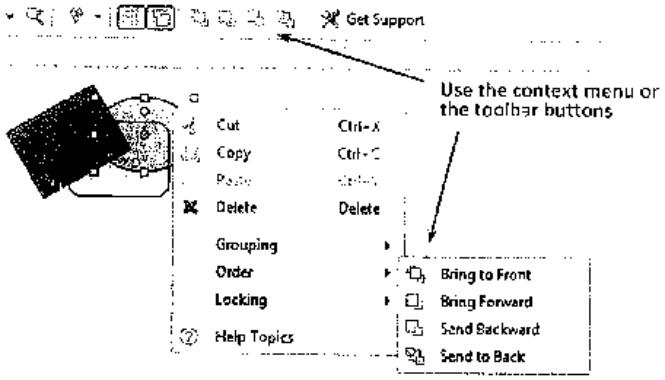


Figure 12.9 Changing the Z-order of a shape

The Z-order of flat shapes defined for 2D animation will not automatically be preserved in 3D animation. Flat shapes located at the same Z level may appear interlaced. To force the Z-order for flat shapes in 3D animation you should assign slightly different Z coordinates for them.

Selecting hidden shapes

Sometimes shapes are hidden below other shapes. You can still select those shapes either by clicking several times on them or by using the **Projects** tree, which shows all elements of the model, including shapes.

▶ **To select a shape that is below other shapes**

1. Click at the position where the hidden shape should be. The top level shape is selected.
2. Keep clicking at the same position until the shape you are looking for gets selected. To find out which shape is currently selected look at the **Properties** view.

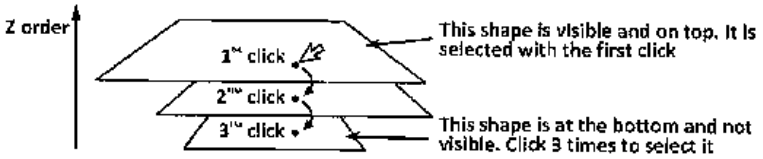

Z order
1st click
2nd click
3rd click
This shape is visible and on top. It is selected with the first click
This shape is at the bottom and not visible. Click 3 times to select it

Figure 12.10 Selecting the hidden shapes by clicking

If you think you have completely lost a shape (or just any object), you can always find it in the **Projects** tree, and Double-click it there to select the object in the graphical editor.

Coordinates and the grid

Graphics in AnyLogic are created in infinite space in the 2D editor. The X axis is horizontal directed to the right, and the Y axis is directed downwards. The Z axis is directed towards you, the viewer. The X and Y axes are shown as grey lines, and when you open the graphical editor the coordinate origin is at the upper left corner of the window. Your graphics as well as the model objects can be located in all four quadrants of the space having positive or negative coordinates. When you run the model, however, the lower right quadrant is shown by default.

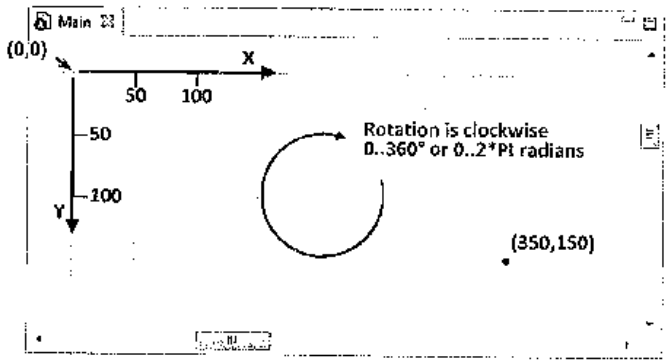


Figure 12.11 Coordinate origin, axes and the grid

The coordinate units correspond to pixels at 100% zoom. To simplify arranging and aligning shapes the graphical editor shows the *grid* and supports *snapping* objects to the grid. At 100% zoom the grid step is 10 and guidelines are shown every 50 pixels. If snap to grid is on, the objects' coordinates and sizes will be equal to N*10. As you change the zoom, the grid step will also change. For example, at 200% zoom the grid step is 5, and at 800% (the maximum) the grid step is 1.

▶ **To change the zoom**

1. Hold the Ctrl button and rotate the mouse wheel, or use the zoom control section of the toolbar

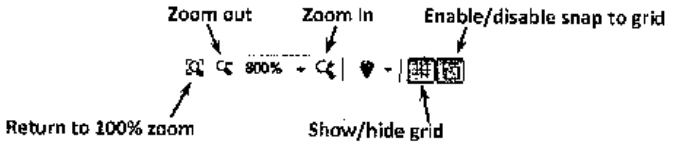


Figure 12.12 Zoom and grid control sections of the toolbar

You can finely adjust the position of your shape without going to large 800% zoom.

▶ **To adjust the position of a shape in between the grid lines**

1. Hold the Shift button and use the Arrow keys to move the shape. The shape coordinates will change by one step regardless of the zoom and grid settings.

The coordinate values in the graphical editor (the design-time coordinates) are integers and correspond to pixels at 100% zoom. The runtime coordinates are real numbers (Java type **double**). Therefore you can position your graphics to within any degree of accuracy by using dynamic properties or the API.

Copying shapes

AnyLogic graphical editor supports cut, copy, and paste operations in the same way any other editor does. You should select the shape(s) first and then use the context menu, toolbar buttons, or main menu commands. You can copy graphics between different active objects or between an experiment and an active object.

There is another way to create a copy bypassing the clipboard: using Ctrl+drag.

▶ **To create a copy of a shape**

1. Select the shape(s)
2. Hold Ctrl button and drag the shape(s) to the new position. The copy is created.

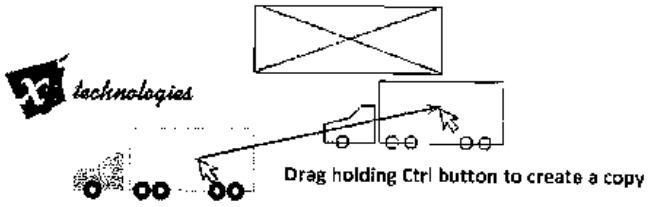


Figure 12.13 Using Ctrl+drag to create a copy of a shape

All these operations (cut, copy, paste via clipboard and copy by Ctrl+dragging) are supported in the Projects tree as well.

Locking shapes – preventing selection by mouse

Graphical editing can sometimes be made easier by desensitizing shapes to mouse clicks to avoid selecting the wrong shape. A simple example is when you have a large background image and wish to draw something on top of it. You can temporarily *lock* shapes for that purpose.

▶ **To lock a shape**

1. Select the shape(s)
2. Choose **Locking | Lock shape(s)** from the context menu

Once the shape has been locked, it will not react to any mouse actions. However, it still can be selected by clicking on its icon in the **Projects** tree.

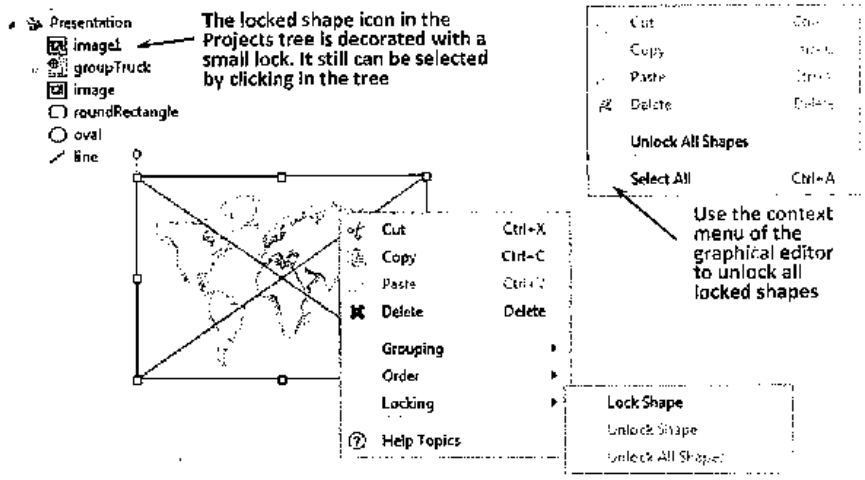


Figure 12.14 Locking and unlocking shapes

▶ **To unlock all locked shapes**

1. Choose **Unlock all shapes** from the context menu of the graphical editor (pops up when you right-click in the empty space of the canvas).

▶ **To unlock a particular shape**

1. Find the shape icon in the **Projects** tree and select it. The locked shape gets selected in the graphical editor as well.
2. Right-click the shape in the graphical editor and choose **Locking | Unlock shape** from the context menu.

General properties of graphical shapes

Any property of any shape can be edited in the **Properties** view. The **General** page contains most frequently used ones: line and fill color, line style and width, etc. It also has a field for the shape name, which actually is the name of the corresponding Java object, and several checkboxes explained in Figure 12.15 below.

The properties specified on the **General** and **Advanced** pages are *static*. Most of them can be overridden by the values typed in the **Dynamic** page or changed using the shape API.

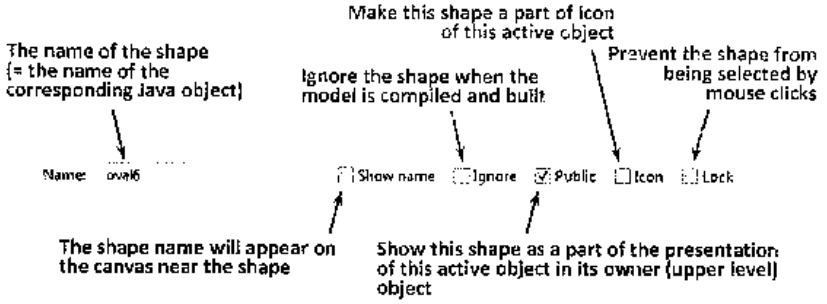

Make this shape a part of icon of this active object
The name of the shape (= the name of the corresponding Java object)
Ignore the shape when the model is compiled and built
Prevent the shape from being selected by mouse clicks
Name: oval6
Show name
Ignore
Public
Icon
Lock
The shape name will appear on the canvas near the shape
Show this shape as a part of the presentation of this active object in its owner (upper level) object

Figure 12.15 General properties common to all shapes

The name of a shape is not normally needed on the canvas. You may only wish to display it when it clarifies your model design. Names of shapes are never displayed during runtime.

Advanced properties of graphical shapes

The **Advanced** page of the properties contains other static (i.e. design time) properties that are used less frequently.

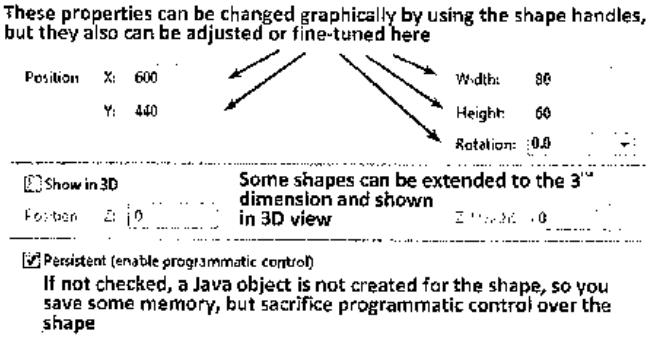

These properties can be changed graphically by using the shape handles, but they also can be adjusted or fine-tuned here
Position X: 600
Y: 440
Width: 80
Height: 60
Rotation: 0.0
Show in 3D
Some shapes can be extended to the 3rd dimension and shown in 3D view
Persistent (enable programmatic control)
If not checked, a Java object is not created for the shape, so you save some memory, but sacrifice programmatic control over the shape

Figure 12.16 Advanced properties of a shape

The position, size, and rotation fields just give you an alternative way of moving and resizing the shape if you do not want to use the graphical editor. The 3D properties are explained in Chapter 14, "3D animation".

The **Persistent** checkbox is checked by default for all shapes and groups. If a shape is *persistent* AnyLogic will create a Java object (like **ShapeOval, ShapeGroup, ShapeImage**, etc.) representing that shape at runtime. In some cases (e.g. when you have an agent based model with very large number of agents that have complex animation) you may wish to optimize your model for memory. So it makes sense to mark the animation shapes of the agents as *non-persistent*. Java objects are not created for non-persistent shapes and they occupy zero memory. However, then you have no programmatic control over such shapes.

12.2. Grouping shapes

Presentation shapes in AnyLogic can be grouped so that you can work with them as if they were a single shape: select, move, resize, rotate, copy, etc. While a shape is a member of a group, it can still be selected individually and edited. At runtime you can dynamically show, hide, rotate, and move grouped shapes by using the group dynamic properties. Groups are also used to provide a new rotation center and coordinate origin for its members.

You can group:

- Simple shapes (rectangles, ovals, polylines, etc), text, and images
- CAD drawings and GIS maps
- Controls (buttons, sliders, text boxes, etc)
- Charts (bar charts, plots, histograms, etc)
- 3D shapes and figures, 3D views, cameras and lights
- Other groups
- Embedded presentations (presentations of embedded active objects)

You cannot group non-presentation elements like events, statecharts, variable and function icons, icons of embedded objects, etc.

▶ **To group one or several shapes:**

1. Select the shapes (by dragging the selection rectangle around or by Ctrl+clicking each shape)
2. Choose **Grouping | Create a group** from the context menu

A group of shapes in AnyLogic has its own "origin", which in general is not the same as the geometrical center of the grouped shapes – it may even be outside the shapes' boundary. The location of the group origin is shown as a small circle with a handle when the group is selected. The group origin serves as rotation center for the group and also as coordinate origin for the shapes – members of the group.

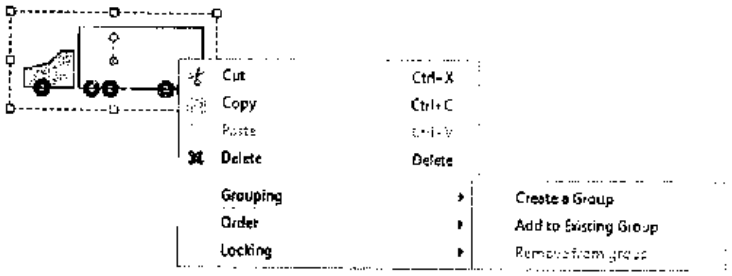


Figure 12.17 Grouping shapes

You may need to adjust the Z-order of shapes after grouping. This is done by selecting individual shapes and bringing them to front or sending them back.

▶ **To add a shape to the existing group**

1. Right-click the shape and choose **Grouping | Add to existing group.** The available groups will be shown as circles with crosses at their origin points.
2. Click the group where you wish to add the shape.

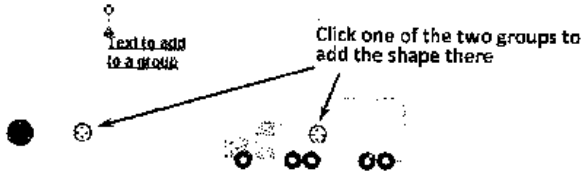


Figure 12.18 Selecting a group to add the shape to

▶ **To select an individual shape that is a member of a group**

1. Select the group
2. Click the shape while the group is selected

Groups in AnyLogic can be nested. If you wish to select the shape that is a member of the inner group, keep clicking the shape until it is selected.

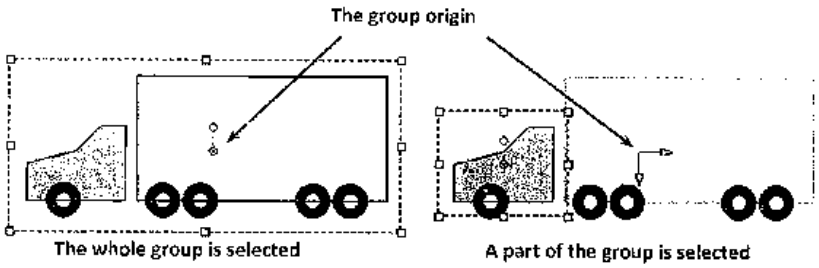


Figure 12.19 A group of shapes in the graphical editor

▶ **To rotate a group:**

1. Pull the handle at the group "origin".

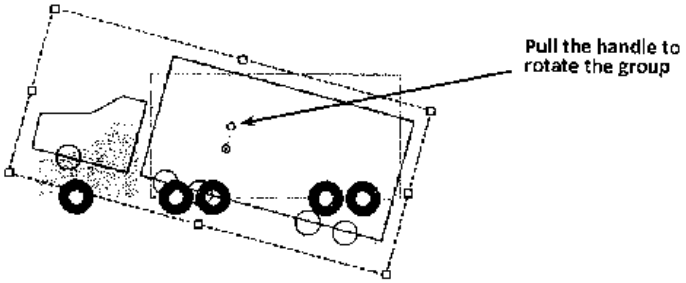


Figure 12.20 Group rotation in the graphical editor

Please note that controls, charts, and 3D views cannot be rotated. If you rotate a group containing such elements, they will only change their location, but orientation will remain the same.

Sometimes the position of the group origin needs to be moved relative to the grouped shapes. This may be needed e.g. to change the rotation and scaling center or coordinate origin of the group.

▶ **To move the group origin relative to the grouped shapes**

1. Select the group.
2. Choose **Select Group Contents** from the group context menu.
3. Move the group contents so that the group origin appears in the right position relative to the shapes.
4. Deselect the group contents by clicking anywhere aside.
5. If needed, select the group again and move it.

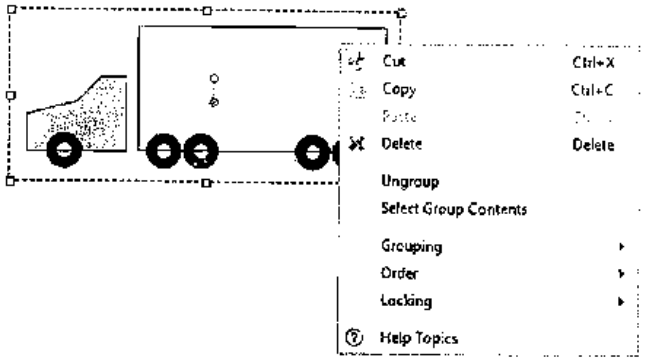


Figure 12.21 Moving the group origin relative to the group contents

A group may be used to set a new rotation point, new scaling center, and new coordinate origin for an individual shape different to the default ones. For example, you may wish to rotate a rectangle around its center, or have its center as its coordinate origin, while the default for both is the upper left corner. You can use the following technique:

▶ **To change the rotation point, the scaling center and coordinate origin of a shape:**

1. Select the shape.
2. Choose **Grouping | Create a group** from its context menu (a new group containing just that shape is created and selected).
3. Choose **Select Group Contents** from the group context menu.
4. Move the group contents (i.e. the shape) so that the group origin appears in the right position relative to the shape.
5. Deselect the shape, reposition and rotate the whole group if needed.

Figure 12.22 Changing the rotation point, scaling center and coordinate origin of a rectangle

Unlike in some other graphical editors such as MS PowerPoint, groups in AnyLogic may be empty containing no shapes or other groups. Such empty groups may be used during runtime to add or remove shapes dynamically. Or they may simply serve as references to particular locations.

▶ **To create a new empty group:**

1. Drag the **Group** element from the **Presentation** palette to the canvas.

While the group is empty, it is shown in the editor as a circle with a cross inside. Once you add the first shape to the group, its icon becomes invisible.

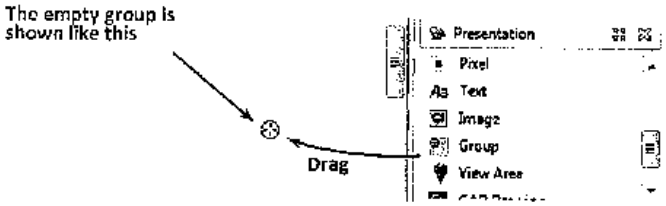


Figure 12.23 Creating an empty group

Pictures that you drag from the **Pictures** palette and drop on the canvas are also regular groups of regular AnyLogic shapes. You can edit them as you like, change their internals, or even ungroup them.

Groups are extensively used at runtime to show, hide, rotate or move parts of the presentation. The group origin serves as the new coordinate origin and rotation center for the group members. For example, if the space where a certain object is going to move has its upper left corner at (150, 200) it makes sense to place the group there and to add the object to the group.

▶ **To define circular movement around the group origin:**

1. Drag **Oval** from the **Presentation** palette to anywhere on the canvas, make it a circle with radius 10, set **Fill color** to **red**.
2. In the **Dynamic** properties page of the circle set **X: 50 * sin(time())** and **Y: 50 * cos(time())**.
3. Run the model. The oval is moving on the orbit around the coordinate origin (the upper left corner of the window).
4. Drag a **Group** from the same palette to the point e.g. (200,150).
5. Click the oval and select **Grouping | Add to Existing Group**, then click the group. When the oval is added to the group its icon is shown as a small circle with a handle
6. Run the model again. The oval is now moving around the group origin.

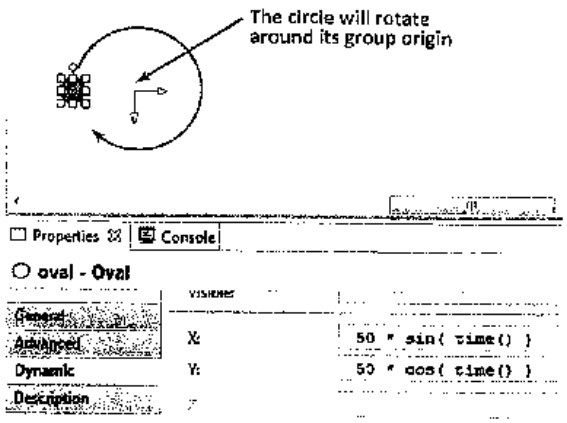


Figure 12.24 Using group as coordinate origin for its member shapes movement

3D Groups

If you wish the group to appear in 3D animation, the group should be marked as 3D group – see **Show in 3D** checkbox in the **Advanced** page of the Group properties. The group will then appear both in 2D and in 3D views.

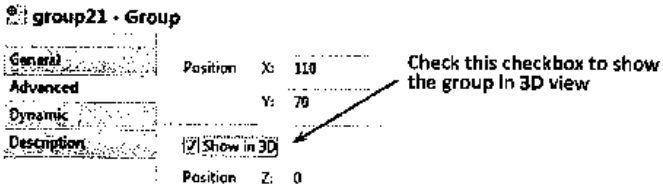


Figure 12.25 Marking a group as 3D group

A 3D group can only contain 3D shapes. If a 3D group contains a 2D shape, the graphical editor will show an error. When several 3D shapes are grouped, the group automatically is marked as 3D. Also, if a group is added to another 3D group, it becomes a 3D group too.

A 3D group is mapped to a class **Shape3DGroup**, which extends **ShapeGroup** and has additional methods like **setZ()**, **getRotationX()**, etc.

Working with the group contents dynamically using API

AnyLogic java class for group is **ShapeGroup** (a subclass of **Shape**). Its API allows you to iterate through the group members and add or remove shapes dynamically. Suppose you have a group with name **group21**. The following code iterates through the contents of the group, finds all ovals and outputs their sizes to the model log:

```
for( int i=0; i<group21.size(); i++ ) {
  Object member = group21.get(i);
  if( member instanceof ShapeOval ) {
    ShapeOval oval = (ShapeOval)member;
    traceln( "Oval Rx: " + oval.getRadiusX() + " Ry: " + oval.getRadiusX() );
  }
}
```

Persistent and non-persistent shapes, as well as replicated shapes, can be mixed in a group. Therefore the method **get()** returns the type **Object** - either an object of a subclass of **Shape**, or an **Integer** – an index of a non-persistent shape, or an object of class **ReplicatedShape**. You need to check (or just cast) the type before doing further work with the group members.

▶ **To dynamically add a shape to the group:**

1. Call `group.add( shape );`

▶ **To dynamically remove a shape from the group:**

1. If you know both the shape and the group, call
   `group.remove( shape );`
   if you know the shape but do not know the group, call
   `shape.getGroup().remove( shape );`

As you can see, a shape knows its group and returns it when you call **getGroup()** method. The top-level shapes (the ones that do not belong to any user-created group) belong to the ever-present default group **presentation** (or **icon**).

On draw extension point – execute custom code on each frame

Groups have an extension point that may be used to do something on each animation frame during the model runtime such as drawing something directly in Java 2D graphics. The extension point is called **On draw**. It is located in the **Dynamic** page of the group properties and offers access to the following variables:

- **Panel panel** – the **JComponent** where the model animation is drawn
- **Graphics2D graphics** – the graphics context used to draw the current frame

The following code typed in the **On draw** field will draw a red circle at (100,100):

```
graphics.setColor( red );
graphics.drawOval( 100, 100, 25, 25 );
```

Groups in the project tree

As with any object in AnyLogic, groups and their contents are shown and can be accessed from the **Projects** tree. The tree shows the hierarchy of the grouped shapes

and their Z-order. Suppose you have drawn the truck as two groups **groupTrailer** and **groupTractor** grouped in the third group **groupTruck**. Then the tree will show the following hierarchy:

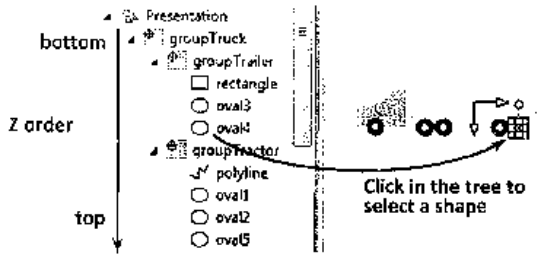


Figure 12.26 Grouped shapes in the Projects tree

Top level groups for active object presentation and icon

Each active object by default has two top-level groups:

- Top level **presentation** group contains all presentation shapes
- Top level **icon** group contains all shapes marked as **Icon**.

These groups may be persistent or not, which may be changed in the **Advanced** page of the active object properties. If they are persistent then **presentation** and **icon** are instances of **ShapeGroup**, otherwise – id's of the non-persistent groups. The cases when these groups are non-persistent are very rare.

12.3. Animation principles. Dynamic properties of shapes

At the elementary level AnyLogic animation is based on the following principles. The graphical shapes have their properties: size, position, rotation, color, image index, text, visibility, etc. These properties can be linked to variables and expressions in the model: values of stocks and flows, states of statecharts, remaining time of events, and so on. When those variables and expressions change their values during the model runtime the shapes change their graphical properties.

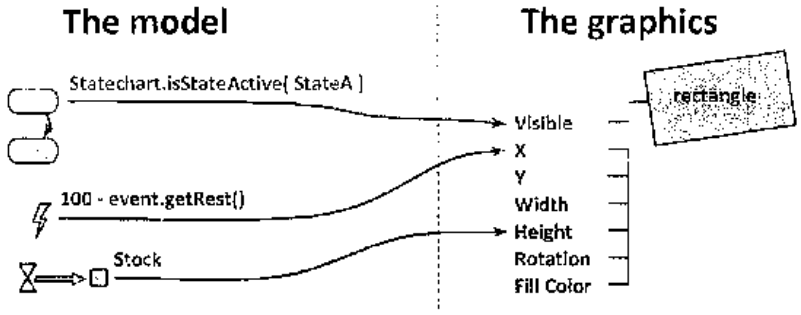


Figure 12.27 The model can control the graphics via dynamic properties of shapes

The agent based modeling framework (see Chapter 3), the Enterprise Library, and other components of AnyLogic offer higher level tools for model animation. For example, an agent can move along a polyline, the waiting entities can be regularly arranged within a rectangle, etc. However, the low-level direct access to the shape properties allows you to achieve very sophisticated custom animation effects.

Dynamic properties of shapes

Linking of the model and the graphical properties is done via *dynamic properties* of shapes located in the **Dynamic** page of the shape's **Properties** view. When you enter an expression in a dynamic property field, the static (i.e. design-time) value of that particular property becomes void at runtime.

In Figure 12.28 you can see the dynamic properties of a rectangle shape. Most of the fields are quite straightforward and do need comments. In some cases you need to know special constants to control the property (e.g. the expression for line style should evaluate to one of those: **LINE_STYLE_SOLID, LINE_STYLE_DASHED** **LINE_STYLE_DOTTED**. You can find these values in *AnyLogic Help* (The AnyLogic Company, 2013). When the expression for line (or fill color) evaluates to **null**, the line (or fill) of the shape is not drawn. The expressions for rotation are interpreted as radians, and you can use the constant **PI**, which equals 180°.

The first dynamic property that is evaluated during animation is Visible. If the expression for Visible evaluates to false, the shape is not drawn at all and all other expressions are not evaluated. Therefore you can specify the expressions in other fields that do not make sense (can raise error) when the shape is not visible and be sure they will not be evaluated.

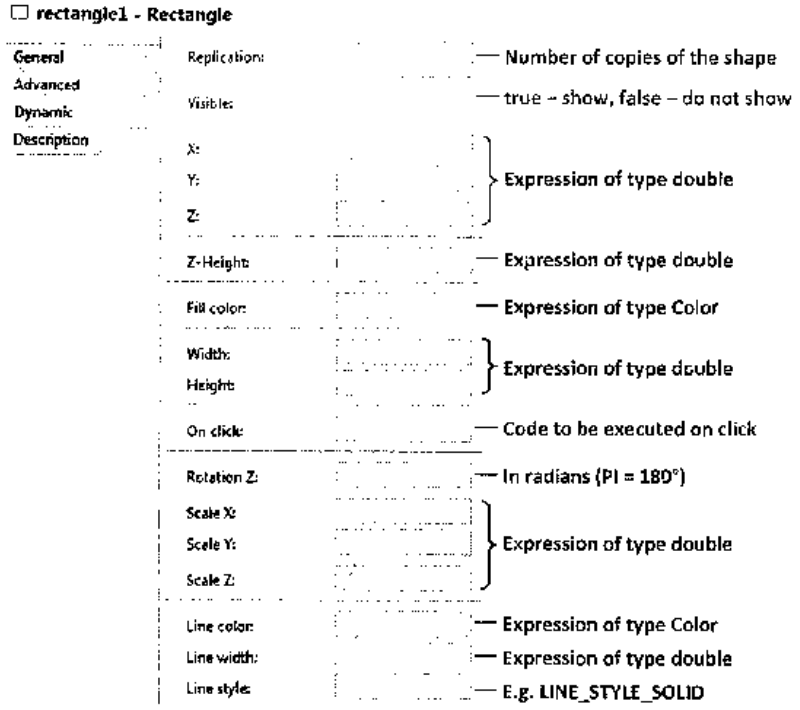


Figure 12.28 Dynamic properties of a rectangle

Example 12.1: Commodity price change animation

We will use a text shape and a little bit of graphics to create a commodity price change indicator. The price itself will be modeled by a variable **price** randomly changing every 1 time unit (e.g. every day). To display the change we need to remember the last day price – another variable **lastDayPrice** will be used for that. The dynamics of the price will be modeled by a cyclic event **dailyChange** (see Section 8.2).

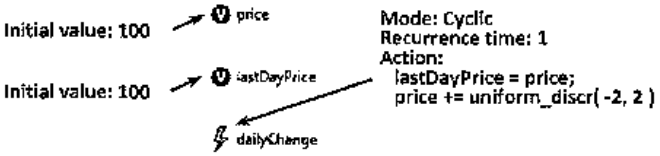


Figure 12.29 A dynamic model of a changing price

▶ **Create the dynamic model of the changing price:**

1. Create a variable **price** by dragging the **Variable** object from the **General** palette and setting its name to **price**.
2. In the properties of the variable set its initial value to **100**.
3. Ctrl+drag the **price** variable to create its copy below it. Change the name of the copy to **lastDayPrice** and leave the initial value unchanged.
4. Create an event **dailyChange** by dragging the **Event** object from the **General** palette and changing its name.
5. Change the **Mode** of the event to **Cyclic** and leave the default **Recurrence time** (1).
6. In the **Action** of the event write the following:

```
lastDayPrice = price;
price += uniform_discr( -2, 2 )
```

7. Run the model. Click the price variable to inspect it and switch to chart view.

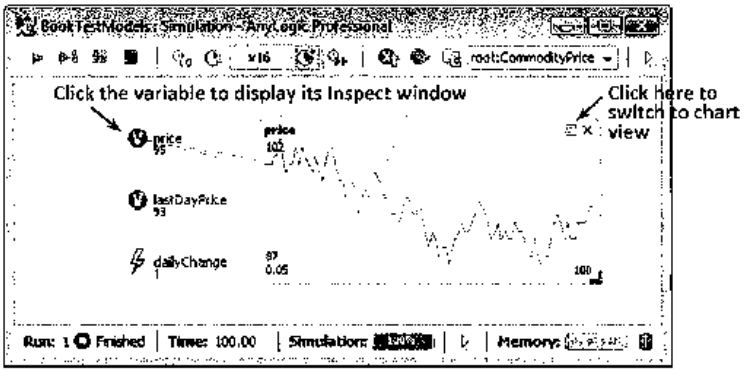


Figure 12.30 The commodity price fluctuations

You should be able to see the price fluctuations. The event **dailyChange** occurs every time unit and does the following: store the current value of the **price** to **lastDayPrice** and adds to the **price** a change value randomly chosen from the set {-2, -1, 0, 1, 2} – this what is returned by the function **uniform_discr(-2, 2)**.

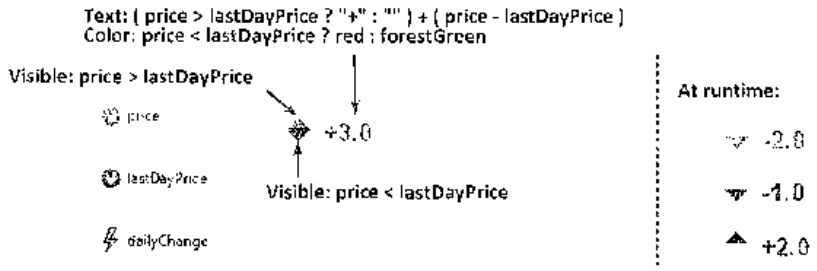


Figure 12.31 The price change animation

▶ **Create the price change animation:**

8. Use polylines to draw two small triangles as shown in Figure 12.31. A triangle is a 3-point polyline with option **Closed polyline** checked in the **General** page of its properties.
9. Choose the red color for the lower triangle and **forestGreen** color for the upper one.
10. In the **Dynamic** page of the triangles properties set the **Visible** property:
    for the red triangle: **price < lastDayPrice**
    for the green triangle: **price > lastDayPrice**
11. Add a text shape on the right of the triangles. Set the text font size to 20 and the font style to Bold. You may write some sample text in the **Text** field of the **General** page, e.g. "+2.0" – just to see how it will look like, the actual text will be set in the **Dynamic** properties page.
12. In the **Dynamic** page of the text properties set:
    Text: (price > lastDayPrice ? "+" : "") + (price - lastDayPrice)
    Color: price < lastDayPrice ? red : forestGreen
13. Run the model.

Depending on the current and the last day prices, either the green triangle will show, or the red one, or, if the price does not change, none. The text color is red if the price goes down, and green otherwise (i.e. if the price goes up or stays the same). The expression for the dynamic value of the text means the following. The part **(price > lastDayPrice ? "+" : "")** evaluates to either string "+" or "" (empty string). The part **(price - lastDayPrice)** obviously evaluates to the numeric value of the price change. If you add a string and a number, the result is a string containing the text representation of the number.

Example 12.2: Elevator doors animation

Let us create an animation of elevator doors. Suppose the dynamics of the doors is modeled by a statechart with four states: **Open, Closed, Opening, Closing**. The doors are

animated as two rectangles. We assume you already know how to draw a statechart with transitions triggered by timeout (see Section 7.3).

▶ **Create the statechart:**

1. Draw a statechart with four states and four transitions as shown in Figure 12.32 below. The name of the statechart should be **doors**.
2. Name the transitions exactly as shown. To let the transition names appear on the screen, check the **Show Name** property of the transitions.
3. All four transitions are triggered by timeouts. Specify the following timeouts:
   StartOpening: 2
   OpeningCompleted: 0.5
   StartClosing: 2
   ClosingCompleted: 0.5

This statechart defines the following behavior of the doors: the doors stay closed for 2 minutes, then open (which takes 30 seconds), then stay open for another 2 minutes, then close again within 30 seconds.

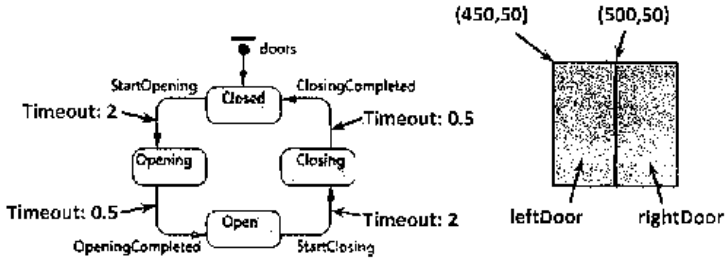


Figure 12.32 The statechart and animation of the elevator doors

▶ **Draw the doors:**

4. Draw the two rectangles as shown in Figure 12.32. The location and sizes are important and should be exactly as shown.
5. Give them the names: **leftDoor** and **rightDoor**.
6. In the **Dynamic** page of **leftDoor** properties enter the following expression in the **Width** field:

```
doors.isStateActive( Closed ) ? 50 :
 ( doors.isStateActive( Open ) ? 0 :
   ( doors.isStateActive( Opening ) ? 50 * OpeningCompleted.getRest()/0.5 :
     50 * ( 1 - ClosingCompleted.getRest()/0.5 )
   )
 )
```

7. In the Dynamic page of rightDoor properties set:
   X : 500 + [50-leftDoor.getWidth()]
   Width: leftDoor.getWidth()
8. Run the model.

The long expression for the width of the left door has the following meaning. If the doors are closed (we test this by testing the current active state of the statechart), the width of the leftDoor is 50. Otherwise, if the doors are completely open (state Open), the width is 0. Otherwise, if the doors are opening, we need to set the door width proportional to the *uncompleted fraction* of opening operation. The fraction is calculated as the time to opening completion (OpeningCompleted.getRest() returns the remaining time before the corresponding transition is taken) divided by the total opening time (0.5). The last possible case is when the doors are being closed. Then the door width is set proportional to the *completed fraction* of operation, which is 1 – uncompleted fraction.

The expression is quite long and we do not want to replicate it for the rightDoor. Moreover, something may change later on and we do not want to correct code in two different places. Instead we will use the fact that the two doors always have the same width, so the width of the rightDoor is set equal to the width of the leftDoor. Note that the right door not only changes its width, but also its X coordinate, hence the expression for X.

If you move the door rectangles you will need to correct the coordinate 500 in the properties, which is undesirable. To create position-independent doors you can group them and use group-relative coordinates. See Section 12.2, "Grouping shapes" for details.

Example 12.3: Stock of money animation

In this example we will visualize a system dynamics stock with a custom graphics. The stock and flow diagram will be very simple: there will be one stock Money, one inflow Income and one outflow Expenses. The graphics will be a bag with a "$" sign on it. The bag will change its size to reflect the value of the stock.

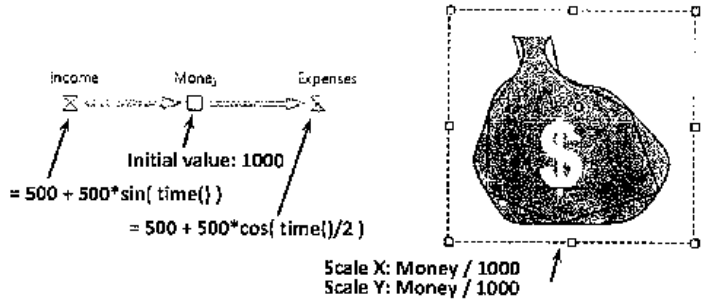


Figure 12.33 Money stock and its custom animation

▶ **Follow these steps:**

1. Draw a system dynamics diagram as shown in the picture.
2. Set the initial value of the **Money** stock to **1000**.
3. Set the formula for
   Income: 500 + 500 * sin(time())
   Expenses: 500 + 500 * cos(time()/2)
4. Draw a curve in the form of a bag. Set the fill color of the curve to e.g. **goldenRod**.
5. Add a text shape on top of the bag. Set its properties:
   Text: $
   Font size: 72
   Font style: Bold
   Color: yellow
6. Select both the bag and the text and group them.
7. In the **Dynamic** page of the group properties set:
   Scale X: Money / 1000
   Scale Y: Money / 1000
8. Run the model.

You should be able to see how the bag changes its size with oscillations of the stock value. Use the **Money** stock's it's inspect window, as shown in Figure 12.34, to view its time chart.

? The bag expands and contracts to/from its center. How could we change the bag's animation to appear to lay flat on a table and only grow up and sidewise as it filled?

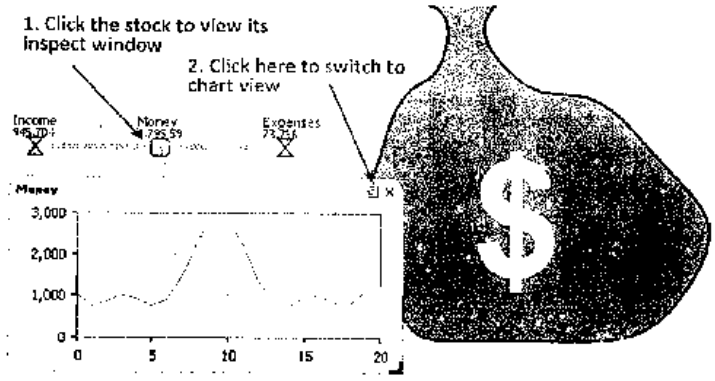


Figure 12.34 The money stock animation

Example 12.4: Missile attack animation

In this example we will use two different images to animate the missile during its flight and after explosion at the target. We will also add a visual effect of explosion purely on animation level. Please prepare two images: one of a missile, another of explosion. It is better if they have the same size or at least the same proportions.

▶ **Follow these steps:**

1. Create an event **hitTarget**. It should be a timeout event that occurs once at time **50**.
2. Add an image object at **(100,150)**
3. Add two images to the image object – the first one with a missile, the second one with explosion. You may or may not use **Original size** option. The image resulting size should be approximately 100 x 100 pixels.
4. In the **Dynamic** page of the image properties set
   Image index: hitTarget.isActive() ? 0 : 1.
5. Right-click the image and choose **Grouping | Create a group.**
6. In the **Dynamic** page of the *group* properties set:
   X: 500 * (hitTarget.isActive() ? (1 - hitTarget.getRest() / 50) : 1)
   Scale X: hitTarget.isActive() ? 1 : 1 + (time() - 50) / 10
   Scale Y: group.getScaleX()
7. Draw the "target" – a thick vertical line from **(550,50)** to **(550, 250)**
8. Run the model.

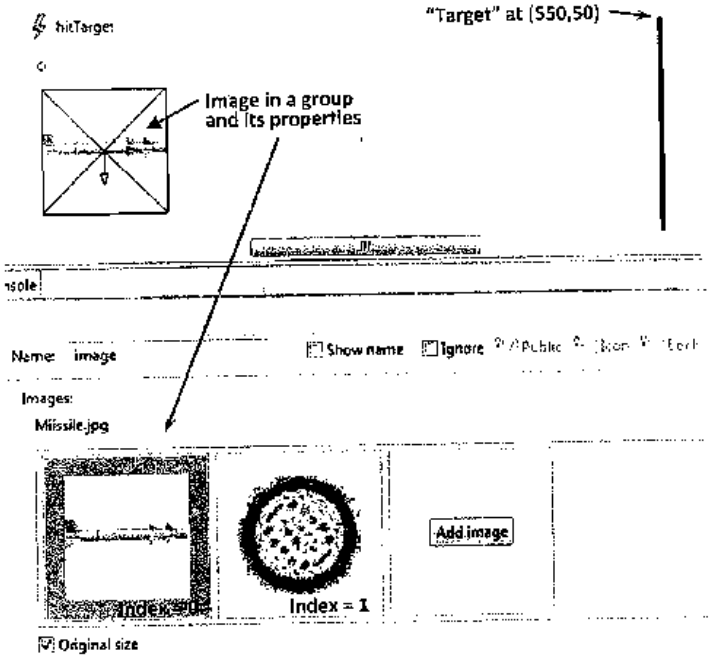


Figure 12.35 Animating missile and explosion with images

The **Image Index** dynamic property is set to change the image of the missile to the image of the explosion once the missile hits the target – and the **hitTarget** event becomes inactive. Grouping was used to shift the scaling center of the image from its upper left corner to its geometrical center. The flight of the missile is animated by the dynamic **x** property of the group: while **hitTarget** event is active the missile X coordinate is proportional to the fraction of the distance to target (at 500) covered by the missile. After the hit, the group stays at X = 500. Finally, the explosion effect is modeled by enlarging the scale of the group after the hit from 1 to infinity. X and Y scales are the same, so scale Y just refers to scale X.

As you may have noticed, the visual effects in this example are achieved solely by setting the shapes' dynamic properties. At the model level we have just one timeout event that occurs once at time 50. Models designed this way are very efficient because when the animation if off in fast execution modes such as during optimization or parameter variation experiments, no CPU time is spent on animation-related tasks.

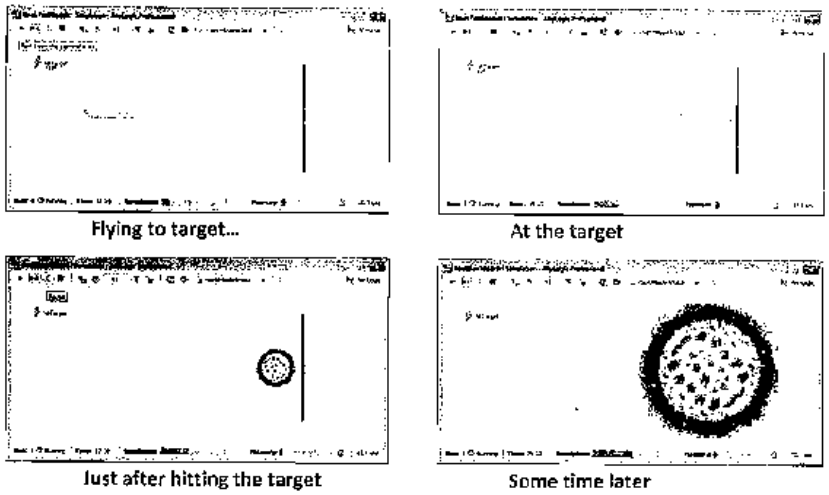


Figure 12.36 Animating missile and explosion with images

Animation frames

2D animation is drawn in *frames*. At the beginning of each frame AnyLogic clears the screen and starts drawing shapes one by one according to their Z-order (see Section 12.1) starting from the bottom. When a shape is drawn, its dynamic properties override its static properties. The property **Visible** is evaluated first, and if it evaluates to false, the shape is not drawn and its other properties are not evaluated.

After the frame is drawn, AnyLogic lets the CPU to continue the simulation in between this frame and the time when the next frame is drawn.. As a result there is a fragment of the simulation executed between frames. Some values in the model may change and be reflected in the next frame (through the dynamic properties of shapes).

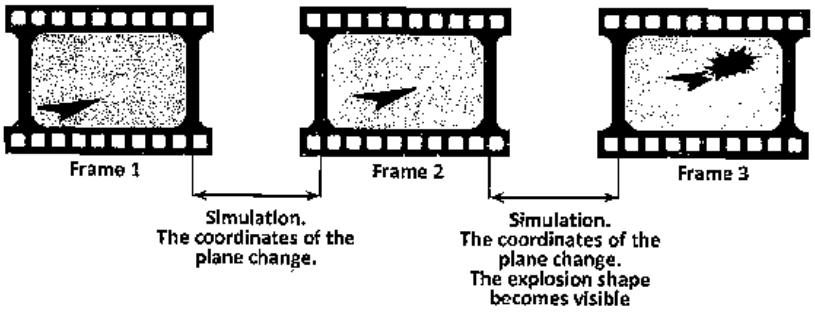


Figure 12.37 Animation frames interlaced with the model simulation

Both frame drawing and simulation consume CPU time, so there is tradeoff between simulation speed and animation smoothness represented by the *frame rate*. You can read more about this in the section about performance of simulation models.

12.4. Replicated shapes

AnyLogic gives you the ability to display an arbitrary (and dynamically changing) number of similar shapes. You need to draw a shape in the graphical editor once and declare it as *replicated shape*. Then you can specify individual property values for every copy of the shape. This is useful for both drawing static regular structures and for animating dynamic collections of similar objects.

Agents (and replicated active objects in general), entities and resource units in the Enterprise Library have their own higher-level animation capabilities any typically you would not use replicated shapes to animate them.

Do not confuse *replicated shapes* with replicated active objects or simulation replications – these are completely unrelated items.

▶ **To declare a shape as replicated**

1. In the **Dynamic** page of the shape properties type an expression in the **Replication** field.

The expression should return an integer number. It can be constant or variable. At runtime AnyLogic will evaluate the expression and draw the corresponding number of copies of the shape. Of course, replication only makes sense if you are able to specify different properties for different copies (otherwise all copies will be identical and drawn one above the other). Once you enter something in the **Replication** field the variable **index** becomes available in all other fields of the **Dynamic** properties page. You can check this by placing the cursor into any other field and then pointing your mouse to the little light bulb on the left.

▶ **To specify different property values for different copies of a replicated shape:**

1. Enter the expression that depends on the **index** variable in the desired dynamic property field.

When drawing the shape copies, AnyLogic will be evaluating the dynamic properties for each copy substituting **index** with 0, 1, 2, ... [Replication-1]. If a property depends on **Index**, it will have different value for different copies.

The property **Visible** is evaluated *before* the property **Replication**. If **Visible** evaluates to false, the shape is not drawn at all regardless the value of **Replication**. Therefore it is not possible to use **Visible** to display some copies of the shape while hiding the

others. To achieve this effect you can set the line and fill colors to **null** for the shape copies that need to be hidden.

Example 12.5: Drawing seats in a movie theater

We will use replication to draw seats arranged in a rectangular movie theater. We will draw the seat once and then we will replicate it. Different copies of the seat will get different coordinates. In this example the replication will result in a static picture. Later on we will extend this example to include animation of dynamic seat sales.

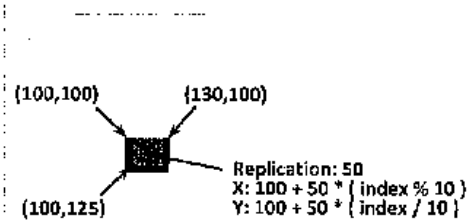


Figure 12.38 A movie theater seat animation – a replicated polyline

▶ **Draw a seat animation**

1. Draw an open three-point polyline as shown in Figure 12.38.
2. Set the polyline line width to 6 pt by entering "6" in the text box.
3. Set the line color of the polyline to **teal** and the fill color to **limeGreen**.
4. In the Dynamic property page set:
   Replication: 50
   X: 100 + 50 * (index % 10)
   Y: 100 + 50 * (index / 10)
5. Run the model.

You should see the 50 seats arranged in 5 rows, 10 seats per row. The number **50** you entered in the **Replication** field is the number of seat copies. By providing X and Y coordinates depending on the seat **index** you have arranged seats in rows. The **index** is varied from 0 to 49. The expression **index % 10** is the remainder of division of **index** by **10**, i.e. it will take the values 0, 1, ..., 8, 9, 0, 1, ... 8, 9, etc. We will use that as the *seat number in a row*, hence the X coordinate is the seat number multiplied by **50** to provide some spacing. The **100** is the coordinate of the first seat. Similarly, the expression **index / 10** is the integer division, and it will evaluate to 0 for seats 0-9, to 1 for the seats 10-19, etc. We will use that as a *row number*. For more information on Java arithmetic see Section 10.5.

The dynamic coordinates override the static values. It does not matter where you have drawn a shape in the editor if you have provided any values in the dynamic coordinate fields. Therefore if we do not add 100 to the dynamic X and Y of the seats, the first seat will be placed at (0,0).

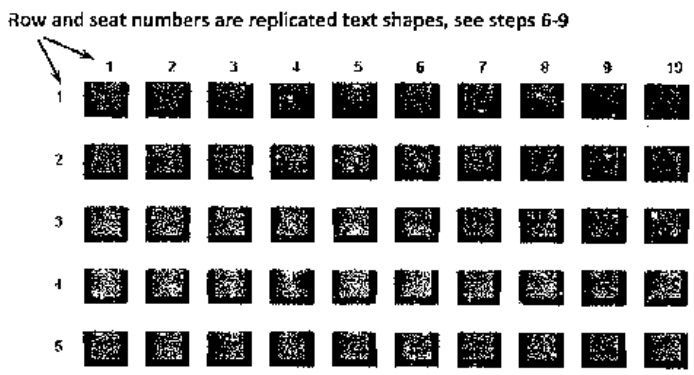


Figure 12.39 Movie theater seats at runtime – 50 instances of a polyline

▶ **Draw seat and row numbers**

1. Create two text shapes, set the static text for both to e.g. "1", and place one above and the other – on the left of the seat polyline, as shown in Figure 12.40.
2. For both texts set the **Font size** to 12, Bold, and the color to **teal**.
3. For the upper text set these dynamic properties:
   Replication: 5
   X: 115 + 50 * index
   Text: 1 + index
4. Set the following dynamic properties for the text on the left:
   Replication: 5
   X: 105 + 50 * index
   Text: 1 + index
5. Run the model.

Figure 12.40 Seat and row numbers – replicated text shapes

For these text shapes we have only set one dynamic coordinate (X or Y), which means that the value for the other coordinate will be the same as the static value, i.e. the coordinate in the graphical editor. As all array and collection numbering in Java starts with 0, we need to add 1 to the **index** to have the row and seat numbers starting with 1.

If AnyLogic detects a numeric value in the dynamic **Text** field, it will automatically convert it to text, so you should not worry about it.

Example 12.6: Selling seats in the movie theater

Now we will make the previous example interactive and show how the properties of a replicated shape can be linked to a dynamically changing collection of values. Imagine the picture you have created appears on the ticket agent's screen at the theater box office.. The unsold seats show in green. The ticket agent clicks on the seats he sells and they become red. In order to display the sold seats we need to remember them. We will use Java array (see Section 10.6) of Boolean values – one value per seat. **true** will mean sold.

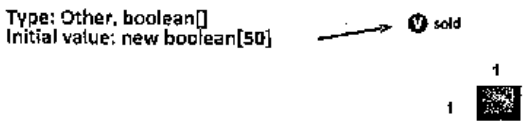


Figure 12.41 Java array of Boolean values

▶ **Follow these steps:**

1. Open the **General** palette and drag the **Variable** on the canvas as shown in Figure 12.41. Give the new variable the name **sold**.
2. In the General property page of the variable **sold** set:
   Type: Other, namely: **boolean[]**
   Initial value: new boolean[50]
3. Select the seat animation (the polyline) and set its dynamic properties:
   Fill color: sold[index] ? red : limeGreen
   On click: sold[index] = true;
4. Run the model.
5. As the model runs, click on some arbitrary seats.

The way you have defined and initialized a Java array of Boolean values in this example is common for all kinds Java arrays: you need to specify the array type in the **Other** type field, and initialize the array with the corresponding newly created construct.

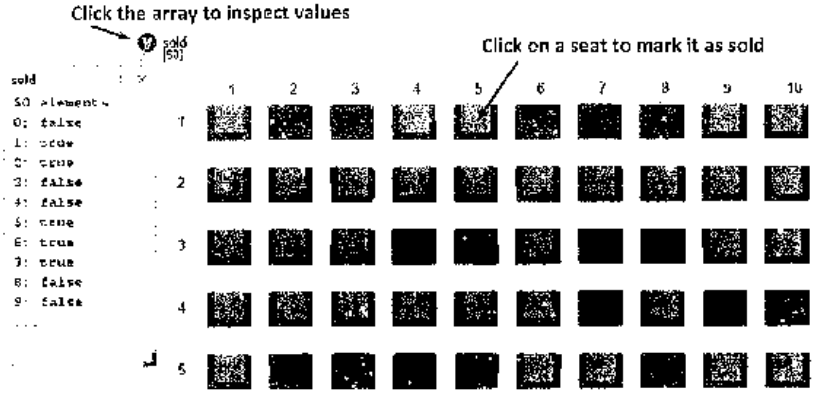


Figure 12.42 Interactive animation of seat selling process

The code you enter in **On click** field gets executed when the user clicks on the shape (see Section 13.3 for more information about handling clicks). The variable **index** is available in this field and it identifies the seat where the user has clicked. The code **sold[index] = true;** sets the value of the array element corresponding to the seat to **true** (by default the Java boolean variables are initialized as **false**). This way the user action through graphics changes the model variable **sold**. And the fill color of the seat dynamically reflects the status (**red** for sold, **limeGreen** for unsold).

Example 12.7: Drawing a flower

This small entertaining example shows how to use replication combined with rotation to arrange shapes in a ring. We will draw a flower petal and then replicate it around the flower center.

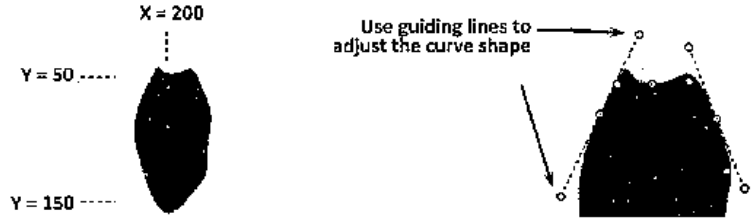


Figure 12.43 A flower petal

▶ **Follow these steps:**

1. Select the **Curve** shape in the drawing mode (see Section 12.1) and draw a flower petal approximately as shown in Figure 12.43. The center of the petal should be at X = 200, and the bottom – slightly lower than Y = 150.
2. Check the **Closed curve** checkbox in the curve properties.

3. Set the line color of the curve to **thistle** and the fill color – to **darkViolet**.
4. Check the **Edit using guiding lines** checkbox in the curve properties.
5. Right-click the curve and choose **Edit points** from the context menu.
6. Use the guiding lines to adjust the curve shape.
7. Right-click the curve and choose **Grouping | Create a group** from the context menu. The group origin appears at the center of the polyline.
8. Drag the group so that its origin is at (200,200).
9. Right-click the group and choose **Select group contents**. The curve (as the only member of the group) gets selected.
10. Move the curve upwards to the position where it originally was drawn. The group origin should remain at (200,200).
11. Draw gold and red circles with the center at (200,200) as shown in Figure 12.44. Send them back so that they appear below the petal.
12. Select the group containing the petal curve and set its dynamic properties:
    Replication: 12
    Rotation: index * PI / 6
13. Run the model.

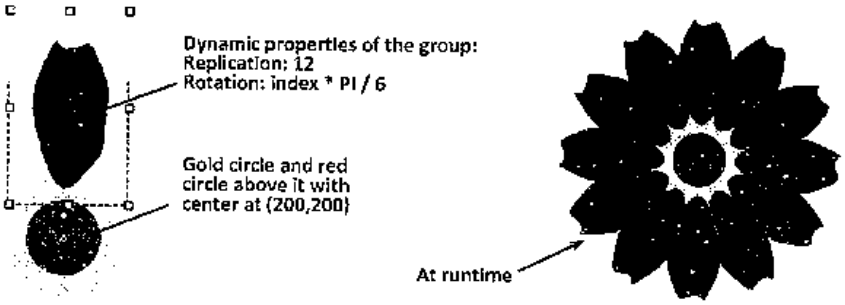

Dynamic properties of the group:
Replication: 12
Rotation: index * PI / 6
Gold circle and red circle above it with center at (200,200)
At runtime

Figure 12.44 Replicating the group with the petal drawing and the resulting runtime picture

You should see a beautiful flower with 12 petals. Grouping in this example was used to provide a new rotation point for the petal shape (this technique is explained in Section 12.2). Each next petal is rotated around (200,200) by the angle **PI / 6** relative to the previous petal.

The copies of a replicated shape are drawn in their natural order, so the last copy always appears on top.

▶ **More fun:**

14. Change the dynamic rotation property of the group to: **index * PI / 6 + time()**.

15. Select the curve (click the curve while its group is selected) and set its dynamic property **Y** to **-40 - 10 * time()** where **-40** is the static Y coordinate of the curve in the group (may be different in your case, see **Advanced** page of the curve properties for the exact value).
16. Run the model.

Example 12.8: Product portfolio bubble chart (BCG chart)

In this example we will use a replicated shape to create a dynamically changing bubble chart, the so-called BCG (Boston Consulting Group) chart. ("Boston Consulting Group", n.d.) It can be used to analyze a company's business units or product lines. We assume that somewhere in the model there is data on the company's products: for each product we know:

- The relative market share
- The business growth rate
- The revenue

In the real model this data may be organized in many different ways, here for simplicity we will use a Java class **Product** with three fields and a collection of products. The dynamics of the model will be implemented by a cyclic event (see Section 8.2) that will add new products and modify the attributes of the existing ones.

▶ **Create the chart frame and axis labels**

1. Use the **Rectangle** shape to draw a square with the upper left corner at (250,50) and size 400x400.
2. Set the **Fill color** of the rectangle to **No fill**.
3. Draw two lines to divide the square into four equal parts as shown in Figure 12.45.
4. Use **Text** shapes to write the axis labels as shown. The text has font size 16 Bold. Use the rotation handle to rotate the labels for the vertical axis by 90°.

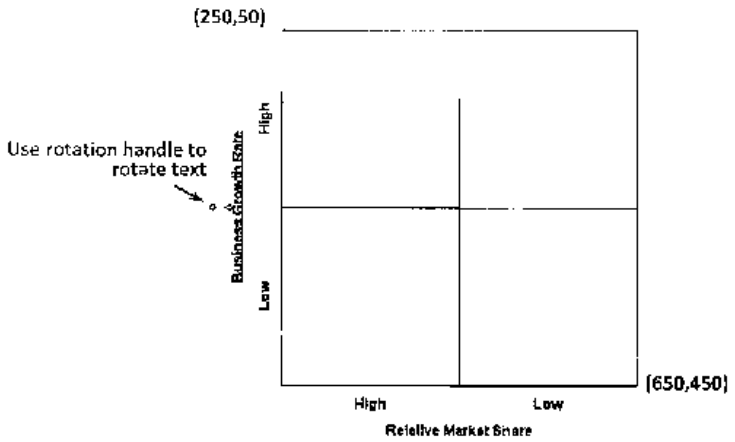


Figure 12.45 The frame and axes of the BCG bubble chart

▶ **Create a Java class Product and a collection of products**

5. Right-click the model (top level) item in the **Projects** tree and choose **New | Java class** from the context menu.
6. Type the name of the class: **Product** and click **Next.**
7. Enter the three fields of the Product:
   RelativeMarketShare of type **double**
   BusinessGrowthRate of type **double**
   Revenue of type double
8. Click **Finish** to close the wizard and create the Java class. The text editor for the class opens, but we will not need it in this example, so you may close it.
9. In the editor of your active object where you have drawn a chart frame and axes, create a new collection by dragging it from the **General** palette to the position (50,100).
10. Type the name of the collection: **products** and specify the **Element class: Product** (this class should be available in the drop-down list).

▶ **Create a cyclic event that will add and modify products**

11. Drag the **Event** from the **General** palette to the position (50,50).
12. Set the event **Mode** to **Cyclic.**
13. Type the following code in the Action of the event:

```
for( Product p : products )
  p.Revenue *= uniform( 0.9, 1.1 );
if( randomTrue(0.2) )
  products.add(
    new Product( uniform( 0.3, 0.9 ), uniform( 0.2, 0.8 ), uniform( 10, 30 ) )
  );
```

14. Run the model and click the **products** collection to inspect it.

The Java class **Product** created in steps 5-8 contains three attributes of the company's product needed to display it in the BCG chart. The event created in steps 11-14 occurs every time unit and does the following: randomly modifies the revenue of each product (the "for loop", see Section 10.8) and, with probability of 20%, adds a new product to the portfolio with randomly chosen attributes. This is of course not a realistic model of a portfolio, but it is suitable for our needs: it can serve as a dynamically changing collection that we will visualize using a replicated shape.

▶ **Create a replicated bubble:**

15. Drag **Oval** from the **Presentation** palette somewhere in the chart boundaries and change its shape to a circle – this will be our bubble.
16. Set both line and fill color of the circle to **mediumPurple**, the change the transparency (see Section 12.6) of the fill color to approximately 100.
17. Set the dynamic properties of the circle:
    Replication: products.size()
    Radius X: products.get(index).Revenue
    Radius Y: products.get(index).Revenue
    X: 400 – 400 * products.get(index).RelativeMarketShare
    Y: 400 – 400 * products.get(index).BusinessGrowthRate
18. Right-click the circle and choose **Grouping | Create a group**. The origin of the newly created group is at the center of the oval.
19. Move the group containing the circle to the upper left corner of the chart, i.e. to (250,50).
20. Run the model.

The number of the bubble copies equals the current number of products and will change as new products are added or existing products are deleted. The size of the bubble is set to be proportional to the **Revenue** of the corresponding product. The X coordinate of the bubble is proportional to the **RelativeMarketShare** so that product with low market share will be on the right, and products with high market share on the left. Similarly, the Y coordinate reflects the **BusinessGrowthRate**. The group is used to provide a new coordinate origin for the oval – the upper left corner of the chart. The construction chart frame + axis labels + grouped bubble is now position-independent and can be freely moved around without the need to change the code. At runtime you should be able to see the bubble chart with an increasing number of bubbles.

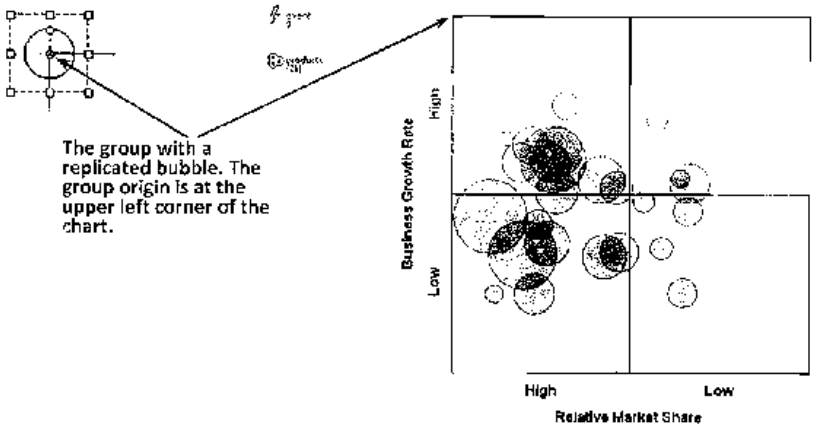

The group with a replicated bubble. The group origin is at the upper left corner of the chart.
Business Growth Rate
High
Low
High
Low
Relative Market Share

Figure 12.46 The BCG bubble chart at runtime

12.5. Shapes' API

Drawing shapes manually using the AnyLogic graphical editor is not the only way to create model graphics, and using shapes' dynamic properties and replication is not the only way to control the graphics at runtime. AnyLogic offers you a rich API to access the shape properties, change, group, and ungroup shapes, create new and delete existing shapes.

Any persistent shape (and by default all Anylogic shapes and groups are persistent) is mapped by AnyLogic to a Java object, which exposes its methods to the user. For example, a rectangle shape is an instance of class **ShapeRectangle** and has methods like **getX()**, **getHeight()**, **setFillColor(Color c)**, **contains(double px, double py)**, etc. A group is an instance of class **ShapeGroup** with methods **add(Shape s)**, **remove(Shape s)**, **size()**, **get(int index)**, and so on. Below you will find some examples of using the shape API.

In some cases you can achieve the same effect by using either dynamic properties of the shape or the shape API. You should choose whichever way seems more natural and elegant, producing less code or producing cleaner code. For example, if the expression in the dynamic properties is too large because several alternative states are checked, it may make sense to use the API to change the property of the shape explicitly when the state changes.

Example 12.9: Using color to show the current state of a statechart

We will use the method **setFillColor()** to change the color of a shape when the state of a statechart changes. This technique is frequently used (e.g. when you develop

animations of agents in agent based models). In this example the statechart will model a consumer and will have three states: **Addressable, OurClient, CompetitorsClient**. We will draw a simple animation of a consumer and change its color from the entry actions of the states.

▶ **Follow these steps:**

1. Use a curve to draw a picture of a consumer, e.g. like the one in Figure 12.47. You may use the guiding lines to adjust the shape.
2. Set the fill color of the curve to e.g. **silver**, and the line color - to **No line**.
3. Draw a statechart with three states as shown.
4. Set the following properties of the transitions:
   Addressable -> branch (decision diamond): **Rate 1.0**
   branch -> OurClient: Condition randomTrue(0.5)
   branch -> CompetitorsClient: Default
   OurClient -> CompetitorsClient: Rate 1.0
   CompetitorsClient -> OurClient: Rate 1.0
5. Specify the following **Entry actions** of the states:
   OurClient: curve.setFillColor(royalBlue);
   CompetitorsClient: curve.setFillColor(orangeRed);
6. Run the model.

You should be able to see how the color of the picture changes from blue to red and back. The model of the consumer is, of course, very simplistic.

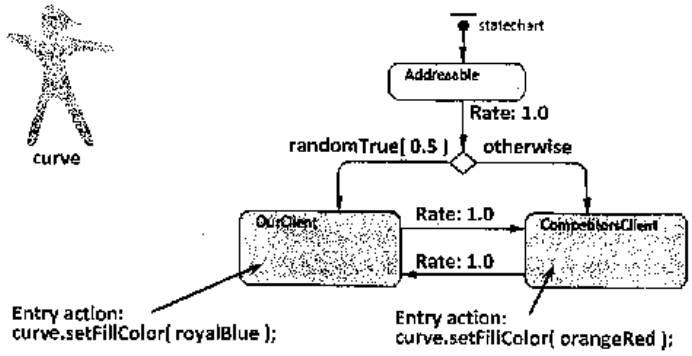


Figure 12.47 The statechart controls the shape's color by calling the shape methods

Example 12.10: Show/hide a callout

In this example we will create a dynamic callout that will be displayed when you click on certain shapes. Such constructs are used to make the models more user-friendly by displaying quick information about certain objects. The callout will be a

group of shapes, for which we will dynamically change coordinates, text content, and visibility.

▶ Draw a callout:

1. Use a polyline tool to draw the callout shape as shown in Figure 12.48. The polyline should be closed.
2. Set the polyline fill color to **khaki**, and line color – to **darkKhaki**.
3. Draw two text shapes on top of the callout as shown. Use the font size 11, Bold. The text that reads "Callout text" is black, and the text "Close" should be blue.
4. Give the text "Callout text" the name **calloutText**.
5. Select the polyline and both texts and create a group. The name of the group should be **callout**.
6. Move the group origin to the sharp end of the callout (see Section 12.2).
7. Set the dynamic property **On click** of the text "Close" to:
   callout.setVisible(false);
8. Run the model. Click the "Close" text on the callout. The callout should disappear.

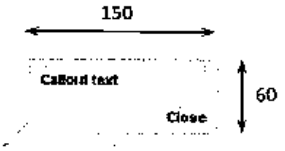


Figure 12.48 The callout

In steps 1-8 we have created the callout group. The group contains the callout background and two text elements. The text **calloutText** will be used later on to dynamically display the information. The text "Close" is click-sensitive and is used to hide the callout by calling its method **setVisible(false)**.

▶ **Draw the shapes that will display the callout:**

9. Move the whole group so that its origin (the sharp end) is at (0,0). You may need to Right-drag the canvas to do it.
10. Draw a red circle e.g. at (100,150). Give it the name **redCircle**.
11. Set the **On click** dynamic property of the **redCircle** to:

```
calloutText.setText( "The red circle" );
callout.setPos( redCircle.getX(), redCircle.getY() );
callout.setVisible( true );
```

12. Ctrl+drag the red circle to create its copy. Set the fill color of the copy to **darkViolet** and the name – to **violetCircle**.
13. In the **On click** dynamic property of the **violetCircle** change the code to:

```
calloutText.setText( "The violet circle" );
callout.setPos( violetCircle.getX(), violetCircle.getY() );
callout.setVisible( true );
```

14. Run the model. Click different circles to display the callout.

Both circles are click-sensitive. When you click on any of them, they dynamically change the text of the **calloutText** shape inside the **callout** group, change the position of the callout and make it visible.

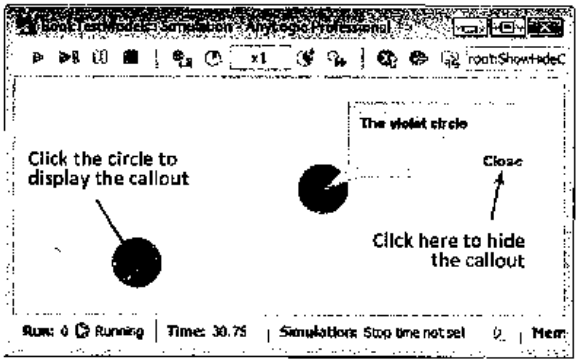


Figure 12.49 The callout

Example 12.11: Read graphics from a text file

Sometimes the graphical configuration of a system is an input of a simulation model. The model reads it upon startup and uses it during execution. In this example we will show how to read graphical data from a text file and create the corresponding shapes in the model. We will use a very simple syntax: each line will contain the shape type (**LINE** or **CIRCLE**), position, and size. The file can look like this:

```
LINE 250 100 150 150
LINE 250 100 300 200
LINE 400 150 250 100
LINE 400 150 350 50
LINE 400 150 550 100
LINE 550 100 650 50
LINE 550 100 600 150
CIRCLE 400 150 30
CIRCLE 250 100 20
CIRCLE 300 200 10
CIRCLE 150 150 10
CIRCLE 350 50 10
CIRCLE 550 100 20
CIRCLE 650 50 10
CIRCLE 600 150 10
```

▶ **Follow these steps:**

1. Use any text editor (e.g. Notepad) and copy the shape description above to a new text file.
2. Save the file under the name e.g. TextFileWithGraphics.txt in the same folder as the .alp model file.
3. In AnyLogic open the **Connectivity** palette and drag the **Text file** object to the canvas at approximately (50,50).
4. In the properties of a newly created **file** object set:
   File name: TextFileWithGraphics.txt
   Mode: Read
   Separators: Line separator and Space.
5. Click the canvas to display the active object properties. On the **General** page enter the following code in the **Startup code** section:

```
while( file.canReadMore() ) {
  String shape = file.readString();
  if( shape.equals( "LINE" ) ) {
    ShapeLine line = new ShapeLine();
    line.setColor( dodgerBlue );
    line.setLineWidth( 4 );
    line.setX( file.readInt() );
    line.setY( file.readInt() );
    line.setEndX( file.readInt() );
    line.setEndY( file.readInt() );
    presentation.add( line );
  } else if( shape.equals( "CIRCLE" ) ) {
    ShapeOval oval = new ShapeOval();
    oval.setFillColor( white );
    oval.setLineColor( dodgerBlue );
    oval.setLineWidth( 4 );
    oval.setX( file.readInt() );
    oval.setY( file.readInt() );
    int r = file.readInt();
    oval.setRadius( r );
    presentation.add( oval );
  }
}
```

6. Run the model.

The **Text file** element in AnyLogic (see Section 11.1) provides an easy-to-use interface to text files. In our syntax, the elements (strings and numbers) are separated with spaces and line breaks so you need to check the corresponding items in the text file properties. The **Startup code** of the active object is called when the active object is created and before it starts execution, therefore, it is a good place to read the input parameters.

Let's review what the code you have just written does. . It reads the text file token by token until the end of file. The first token is assumed to be a string "LINE" or "CIRCLE". If it is a line, a new **ShapeLine** is created, its color is set to **dodgerBlue**, and line width to 4. Then the coordinates of the line ends are set to the values of the next four tokens read from the file, which are assumed to be integer numbers. Finally, we need to add the newly created line to the presentation of our model, which is done by calling **presentation.add(line)**. Circles are treated in the same way.

The text file reading code in this example does not handle the input file errors. If the file syntax differs from the assumed, a read error will be signaled and the model will terminate. You may consider adding some code to ignore the wrong tokens or to give better error diagnostics.

When the model is run, the picture should look like the one in Figure 12.50. Please note that as long as the lines precede the circles in the input file the circles are created later and are drawn on top of the lines.

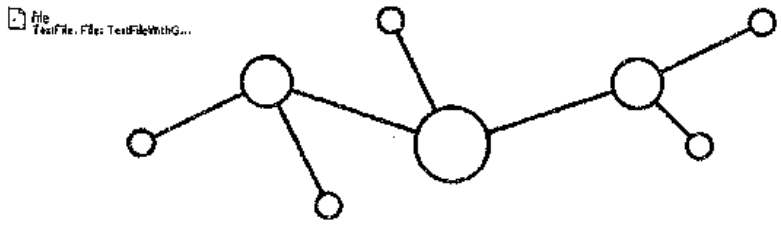


Figure 12.50 The graphics created from the input text file

Example 12.12: Find all red circles

We will look through all shapes at the top-level group of the model presentation, find all red circles and add them to a collection. The technique may be useful to configure the model according to the graphics drawn by the user.

▶ **Follow these steps:**

1. Draw some arbitrary picture on the canvas using various shapes. In that picture create several red circles.
2. Drag a **Collection** object from the **General** palette to the canvas. Give it the name **redCircles**.
3. In the properties of the collection set the **Elements class** to **ShapeOval**.
4. Click the canvas to display the active object properties. On the **General** page enter the following code in the **Startup code** section:

```
for( int i=0; i<presentation.size(); i++ ) {
```

```
Object o = presentation.get(i);
if( o instanceof ShapeOval ) {
  ShapeOval oval = (ShapeOval)o;
  if( oval.getFillColor() == red )
    redCircles.add( oval );
 }
}
```

5. Run the model and click the **redCircles** collection to inspect it.

The startup code gets executed after the active object and all of its graphics are created so you can access the shapes there. The code iterates through all shapes of the top-level group presentation. Every shape is tested to be a circle and, if yes, to have red fill color. All red circles are added to the collection, which contains elements of class **ShapeOval** (this is AnyLogic Java class for circles and ellipses).

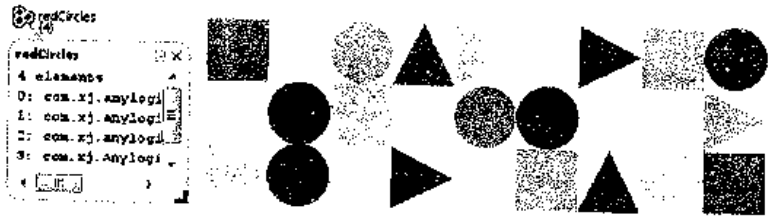


Figure 12.51 The four red circles have been found and added to the collection

Example 12.13: Resize the red circles

Now we will extend the previous example by adding two buttons that will change the size of the circles.

▶ **Follow these steps:**

1. Open the **Controls** palette and drag the **Button** object to the canvas.
2. Change the **Label** of the button to "+".
3. In the **Action** field of the button properties write:

```
for( ShapeOval circle : redCircles ) {
  circle.setRadiusX( circle.getRadiusX() * 1.1 );
  circle.setRadiusY( circle.getRadiusY() * 1.1 );
}
```

4. Ctrl+drag the button to create another similar button below it. Change the label of the second button to "-".
5. Modify the code of the second button: replace "1.1" by "0.9".
6. Run the model. Try to press "+" and "-" buttons.

In the buttons action code the sizes of all red circles in the collection are increased or decreased by 10%. The **get...** and **set...** methods of the **ShapeOval** are used.

API of non-persistent shapes

While the regular (persistent) shapes become Java objects and expose their methods to the user, the non-persistent shapes (see Section 12.1) do not have any objects associated with them at runtime. While you can control the non-persistent shapes at runtime using their dynamic properties and replication, you cannot call their methods, create or delete the shapes dynamically, change the content of the non-persistent groups, etc. However, there are methods of the active object that will return information about the non-persistent shapes, should you need it. Names of non-persistent shapes become integer ids, and those methods require the shape id as an argument. For example, to obtain the X coordinate of the non-persistent rectangle shape with the name **rectangle21** you should call **getShapeX(rectangle21);** to obtain the fill color of the 10th copy of the replicated non-persistent curve **curve4** you should call **getShapeFillColor(curve4, 9)** (remember that numbering in Java starts with 0, hence 9 for the 10th copy).

AnyLogic Java class hierarchy for shapes

The full list of shapes' methods is available in *AnyLogic Classes and Functions* (The AnyLogic Company, 2013). In Figure 12.52 below we give the Java class hierarchy for shapes. The classes in boxes represent actual shapes, the ones without are intermediate classes.

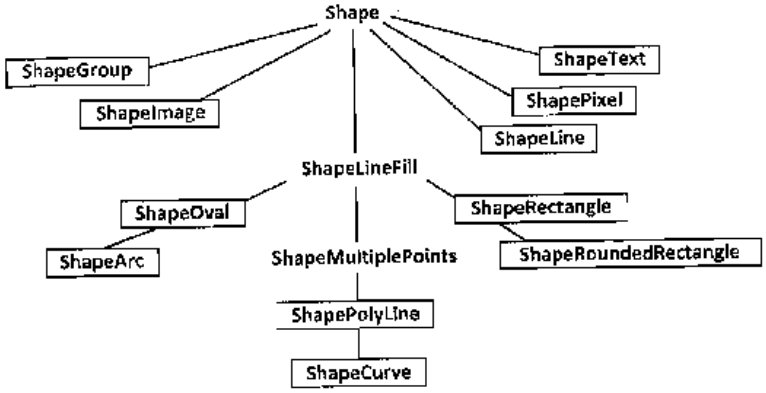


Figure 12.52 AnyLogic Java classes for shapes

12.6. Colors and textures

Static colors and textures of AnyLogic graphical objects (shapes, controls, charts, etc) are set using the **Colors dialog** and **Color picker**. Dynamic colors (those that may override the static colors at runtime) are defined using expressions in the dynamic

properties of shapes (see Section 12.3). You can also set colors using the shape API (see Section 12.5).

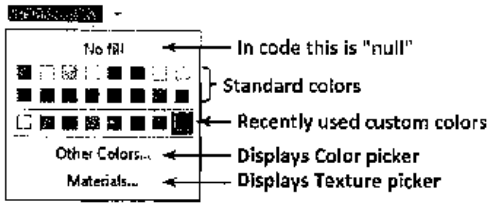


Figure 12.53 Colors dialog

The **Colors dialog** that pops up when you click the button near the color box (see Figure 12.53) contains 16 standard and 8 recently used custom colors. To set a different color or a texture you should choose **Other colors** or **Materials** from the menu.

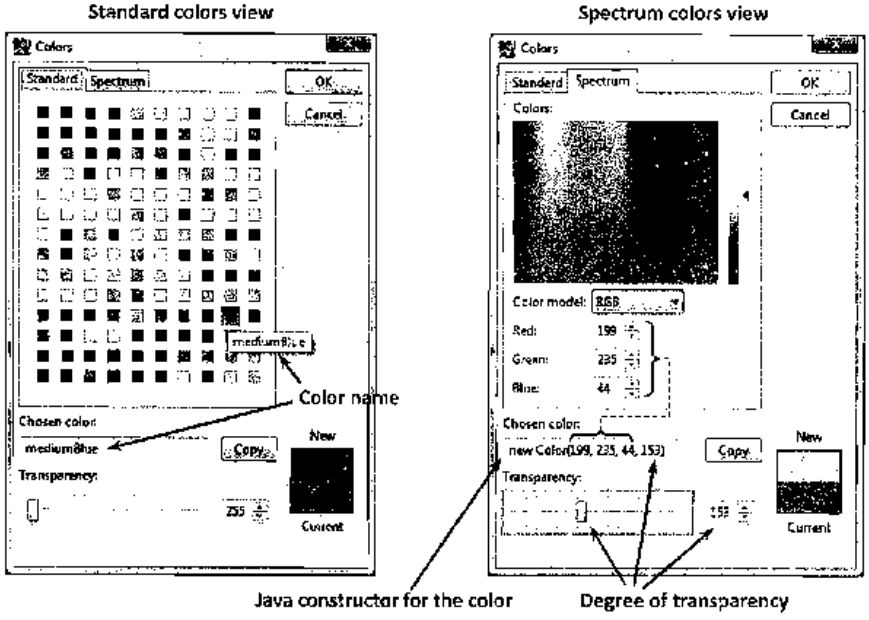


Figure 12.54 AnyLogic color picker

While the color picker saves the chosen color in the shape properties, you can also use it to obtain the Java representation of the color, which you can insert in your expressions and code. The code equivalent for "no color" is null. The standard colors have names like **red, blue, yellow, limeGreen, gold**, etc - these are actually the names of

Java constants of type Color. For the nonstandard colors you should use constructors (see Section 10.2) like new Color(139, 14, 191) or new Color(0, 255, 0, 128). To copy the Java representation of a color press the Copy button nearby.

The word new is needed to tell Java that a new color is created. This happens each time the expression with new... is evaluated. Therefore if you are using constant colors in your code, especially in the dynamic properties of shapes (which are evaluated on every frame), it makes sense to define a constant of type Color and refer to it in the code. The constant will be created once, so the simulation performance improves.

There are several functions that help to manipulate colors. For example, to obtain a darker or brighter version of a particular color color you can call:

```
color.darker()
color.brighter()
```

Color is a standard Java class and you can find more information on these and other methods in Java class reference (Oracle, 2011). In addition, AnyLogic offers some more useful functions, e.g. spectrumColor(), semiTransparent()., and lerpColor().

Example 12.14: Choosing appropriate colors for an arbitrary number of objects

Assume you have several different but similar objects in your model, which you wish to display using different colors that go well together. The function spectrumColor(index, period) will return an attractive color with a given index out of period different colors evenly distributed over the whole spectrum. In the example below 10 (the period) can also be a variable.

▶ **Follow these steps:**

1. Use a polyline to draw a shape, e.g. a star like shown in Figure 12.55.
2. In the dynamic properties of the star set:
   Replication: 10
   x: 100 + index * 50
   Fill color: spectrumColor(index, 10)
3. Run the model.

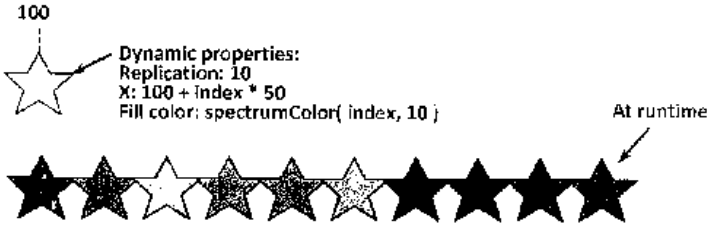


Figure 12.55 Good looking colors obtained using the function spectrumColor()

Transparency

Transparency can be used in many ways to enhance the graphical presentation of your models. For example, a semi-transparent color can show a certain (virtual) area on top of a main map or plan. Dynamically changing degree of transparency can reflect a certain property of a model object, e.g. density, readiness, level of emergency, etc.. If you have a large number of thin lines which display the links between agents (see Section 3.7) they may look better semi-transparent. You can choose a semitransparent color for parts of your histograms, charts or plots, and so on.

Static transparent colors can be set by using the color picker. The **Transparency** slider changes the degree of transparency from 255 (fully opaque) to 0 (fully transparent). If there is any degree of transparency (i.e. the transparency has other value than 255), the fourth parameter is added to the Java color constructor as shown in Figure 12.54.

There is also an easy way of obtaining the semitransparent colors programmatically: you should call the function **semiTransparent(<base color>)**, which produces the color of the same hue and transparency equal to 128.

To control the degree of transparency dynamically you should vary the fourth parameter of the color constructor. For example, you can use the expression like **new Color(255, 0, 0, x)** in the dynamic **Fill color** property of a shape where **x** is an integer that varies between 0 and 255, to obtain semitransparent red colors.

Semitransparent shapes take more time to render than fully opaque shapes.

Example 12.15: Using transparency to show coverage zone

We will use semitransparent circles to show a hypothetical cell phone coverage zones on top of the map of the USA.

▶ **Follow these steps:**

1. Drag the USA map object from the **Pictures** palette to the canvas.

2. Create several circles (object **Oval** in the **Presentation** palette) and place them over the map approximately at the locations of the following cities: New York, Chicago, San Francisco, Los Angeles, Seattle, Orlando, Houston, and Boston.
3. Adjust the sizes of the circles as shown in Figure 12.56.
4. Select all circles (e.g. by choosing **Select All** from the canvas context menu and then deselecting the USA map by Ctrl+clicking it).
5. In the **General** page of the circles' properties set the line color to **No line**.
6. Open the **Color picker** for the fill color.
7. Click the red color (the top right square).
8. Move the **Transparency** slider to the position 100 (you can finely adjust the value by using the Arrow keys).
9. Click **OK**.

You will see the semitransparent circles on top of the map. Note that the color is more intense where the circles overlap.

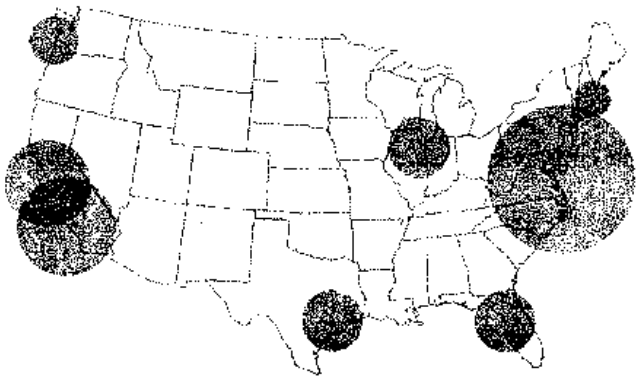


Figure 12.56 Coverage map shown using semitransparent colors

Example 12.16: Show population density using color interpolation

We will use AnyLogic function **lerpColor()**, which performs linear *color interpolation*, to visualize the dynamically changing population density of US states. We will use the USA map that can be found in the **Pictures** palette. The map is a group of state shapes, so we can set the dynamic fill color property for any state individually. We will consider only two states: Texas and California and use system dynamics stock and flow diagram to model the hypothetical migration.

▶ **Follow these steps:**

1. Drag the USA map object from the **Pictures** palette to the canvas.

2. Draw the system dynamics diagram (see Section 5.1) as shown in Figure 12.57. Enter the following values and formulas (the initial values are taken from the (United States Census Bureau, 2013), the migration is of course made up):
   PopulationCA initial value = **36756666**
   LandAreaCA = 155959
   DensityCA = PopulationCA / LandAreaCA
   Migration = PopulationCA / 10
   PopulationTX initial value = **24326974**
   LandAreaTX = 261797
   DensityTX = PopulationTX / LandAreaTX
3. Click the US map (this will select the whole group) and then click California to select the California shape.
4. In the dynamic properties of the California shape set **Fill color** to: **lerpColor(DensityCA/500, white, red)**.
5. Similarly, set the dynamic fill color or the Texas shape to: **lerpColor(DensityTX/500, white, red)**.
6. Run the model.

The system dynamics model assumes 10% of the California population moves to Texas each year, and the population density variables change in the range 0..500 approximately. Let us assign white color to zero density and red color to density 500 persons per square mile. The function **lerpColor(** fraction, **white, red)** returns **white** if fraction <= 0, **red** if fraction >= 1, and a color in between white and red if fraction is between 0 and 1. Therefore we need to divide the density by 500 to get a value between 0 and 1.

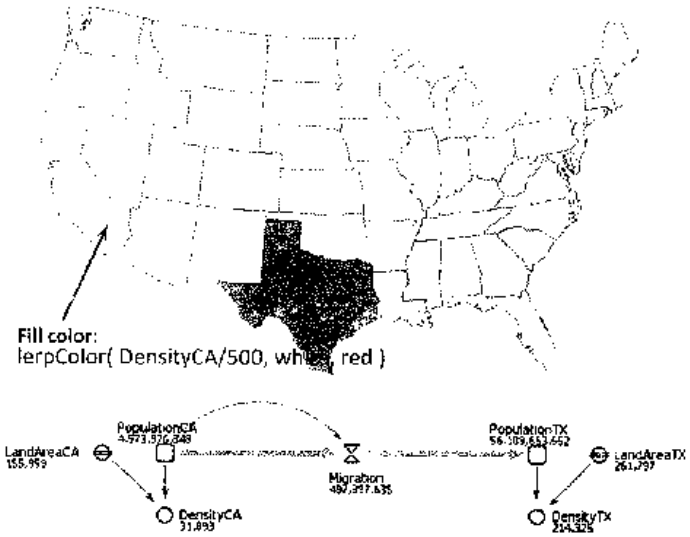


Figure 12.57 The state population density shown by color between white and *red*

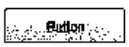
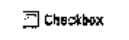
Chapter 13. Designing interactive models: using controls

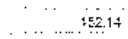
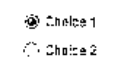
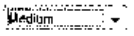
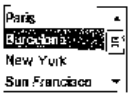
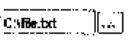
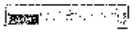
You can make your AnyLogic models interactive by including various *controls* (buttons, sliders, text inputs, etc) into the model front end, and also by defining reactions to mouse clicks. The controls can be used both to set up parameters prior to the model execution and to change the model on-the-fly.

Controls can be found in the **Controls** palette and are created and edited in just the same way as shapes. Controls can be grouped (see Section 12.2) with shapes and other controls and can be replicated. Just like shapes, controls have dynamic properties (see Section 12.3) that can be used to change their size, position, availability, and visibility at runtime. Controls of the embedded object can be set to appear on the container object presentation.

Controls always appear on top of any other graphics (shapes, model objects, etc), regardless of the z-order and grouping. You should avoid overlapping controls as this may produce undesirable visual effects.

Controls that have state or content (such as slider, radio buttons, edit box, etc) in AnyLogic have *value* and can be *linked* to variables and parameters, so that when the user changes the control state, the linked object changes too (but not vice versa). In addition, you can associate an arbitrary action with a control, e.g. call a function, schedule event, send message, stop the model, and so on. The action gets executed each time the user touches the control. The value of the control is typically available as **value** in the control's **Action** code field and also is returned by the control's **getValue()** method. Quick information about each control is given in the Table below.

Control	Type of value	Can be linked to type	Comments
Button 			Is used to perform custom immediate actions in the model. You write code in the Action field and that code gets executed when the user presses the button.
Checkbox 	boolean	boolean	

Edit box 	String	String or any numeric (double, int, etc.)	In addition to linking the edit box to a variable, you may define your own custom code that handles (validates, accepts, rejects) the user's input.
Radio buttons 	int	int	The first choice corresponds to value 0, the second – to 1, and so on.
Slider	double	double or any numeric (int, etc.)	You can define the minimum and maximum values of the slider.
Combo box 	String	String	Can be editable or fixed which limits choices to a defined set.
List box 	String	String	Can work in single or multiple selection modes. If you choose **Multiple selection**, the list box cannot be linked and its value is available via the **getValues()** method that returns the array **String[]**.
File chooser 	String		Displays the system **Open File** or **Save File** dialog and keeps the result as **String** containing the file name with full path. You can define filters based on file extensions.
Progress bar 	double	double or any numeric	In the deterministic mode displays a given progress value. In the nondeterministic mode displays "activity is going on". Both **Progress value** and **Deterministic** properties are dynamic, i.e. constantly evaluated at runtime.

Example 13.1: Slider linked to a model parameter

We will create a parameter and a slider and link them. We will continue this example later and build upon it.

▶ **Follow these steps:**

1. Create a parameter by dragging the **Parameter** object from the **General** palette.
2. In the **General** properties of the parameter set its default value to **50**.

3. Open the **Controls** palette and drag the **Slider** object to the canvas near the parameter. Extend the slider a bit as shown in Figure 13.2.
4. In the **General** properties of the slider check the **Link to** checkbox.
5. Set the following properties:
   Link to: parameter
   Minimum value: 0
   Maximum value: 100
6. Run the model. Move the slider and watch the parameter value (you can click the parameter and display its time chart).

You may have noticed that when the model starts, the slider position is set to the initial value of the parameter (50). If the value of the parameter is outside the range of the slider, the slider will be moved to the closest possible position, but the value of the parameter will not change until you touch the slider.

You may use Arrow keys to make fine adjustments to the slider position.

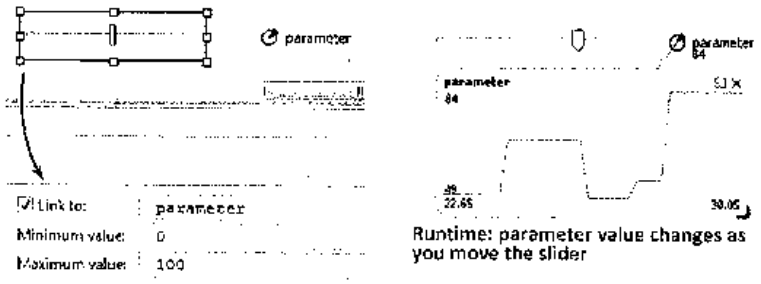


Figure 13.2 Slider linked to a parameter

You should keep in mind that if the value of the parameter is changed "externally", i.e. not by the slider, the slider position will not be automatically adjusted. If you wish the link to always work "both ways" you can, for example, add the corresponding code in the **On change** field of the parameter (see the next example).

Example 13.2: Buttons changing the parameter value

We will add two buttons to the previous example. The buttons will increase and decrease the parameter value by 1, yet keeping the parameter in the range 0-100.

▶ **Add the buttons:**

1. Create two buttons (drag them from the **Controls** palette) and arrange them as shown in Figure 13.3.

2. On the **General** page of the left button properties set:
   Label: -1
   Enabled: parameter >= 1
   Action: set_parameter(parameter-1);
3. On the **General** page of the right buttons properties set:
   Label: +1
   Enabled: parameter <= 99
   Action: set_parameter(parameter+1);
4. Run the model. Move the slider and use buttons.

Do not confuse the *name* of a control with its *label*. The name is the name of the Java object created for the control (and used to access the control's API), and the label is the text that appears near or on the control on the screen. The label can be changed dynamically during runtime.

The (auto-generated) method **set_parameter(parameter-1);** is called in the button action instead of more simple code **parameter--;** to make sure the parameter is changed correctly, in particular its **On change** code is called. The expression you enter in the **Enabled** field of a control is constantly evaluated during runtime, and, if it evaluates to false, the control gets disabled, and vice versa. In our case we do not want the user to be able to drive the parameter value out of range 0-100, so when the value is closer than 1 unit from the range border, we disable the corresponding buttons.

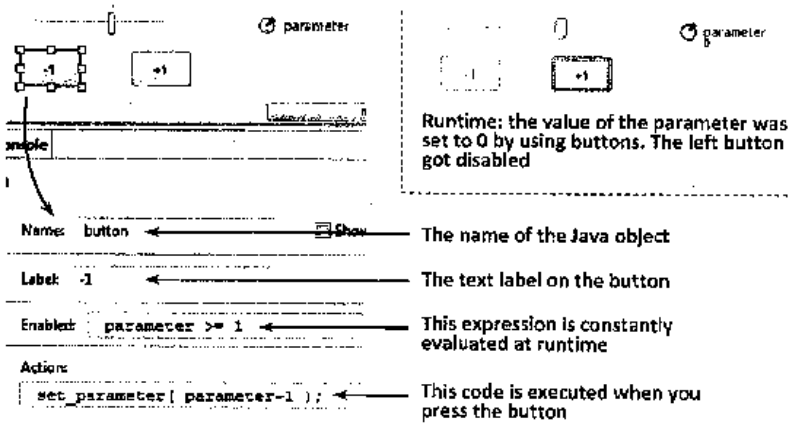

parameter
-1
+1
Runtime: the value of the parameter was set to 0 by using buttons. The left button got disabled
Name: button
Show
The name of the Java object
Label: -1
The text label on the button
Enabled: parameter >= 1
This expression is constantly evaluated at runtime
Action:
set_parameter(parameter-1);
This code is executed when you press the button

Figure 13.3 Buttons incrementing and decrementing the parameter value

As you see, when the parameter changes as a result of you pressing the buttons, the slider stays at its position, which becomes inconsistent with the parameter value.

To fix this we will have the slider move each time the value of the parameter changes.

▶ Add the On change code of the parameter

5. In the **General** page of the parameter properties write the following code in the **On change** field: **slider.setValue(parameter);**
6. Run the model. Press the buttons and watch the slider.

The parameter value, the slider position, and the enabled/disabled buttons are now all consistent.

Example 13.3: Edit box linked to a parameter of embedded object

Not only can controls be linked to the parameters and variables of the "current" active object (i.e. the one where they belong to), but also to the parameters of other objects such as embedded ones. In this example we will link the edit box to a parameter rate of the **Source** object in a simple process model.

▶ **Follow these steps:**

1. Open the **Enterprise Library** palette and drag the **Source** object and the **Sink** object to the canvas.
2. Double-click the output port of **source** and connect it to the input port of **sink**.
3. Open the **Controls** palette and drag the **Edit box**. Drop it on the left of source object.
4. In the **General** properties of the edit box set:
   Link to: checked, value: **source.rate** (use code completion, see Section 10.4)
   Minimum: 0
   Maximum: 100
5. [optional] Create an explanatory text "Arrival rate" on the left of the edit box.
6. Run the model.
7. Click the source object to bring up its inspect window.
8. Try to set different values of the arrival rate: 20, 0, -1, "abc", etc.

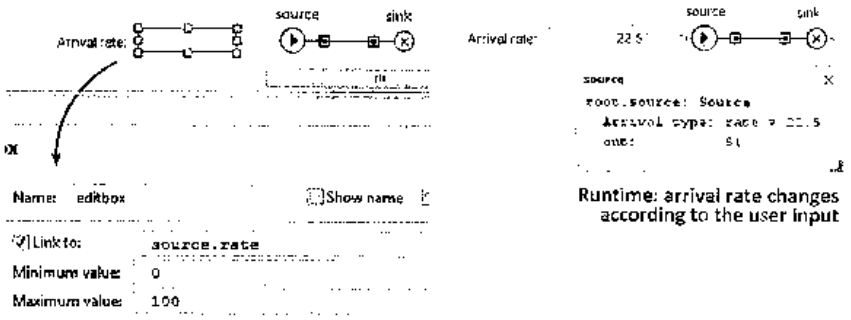


Figure 13.4 Edit box linked to a parameter of embedded Source object

The rate parameter of source changes with each new valid value entered; the corresponding **set_rate()** method is called each time. In case the edit box is linked to a parameter of numeric type, the invalid inputs, as well as numeric inputs that are out of the given range, are automatically rejected and the value does not change.

Example 13.4: Radio buttons changing the view mode

Let us use radio buttons to change the view mode of a model with the US map as part of the interface. Assume we want to give the user the ability to see three possible configurations: a) the map with main cities, b) the map without cities, or c) the cities without the map.

▶ **Follow these steps:**

1. Drag the **US map** from the **Pictures** palette to the canvas.
2. Create a small red circle by dragging it from the **Presentation** palette, adjusting its size to 10 pixels and setting the fill color to red.
3. Place the circle on the pacific coast in southern California, approximately where Los Angeles is located.
4. Zoom in the editor and Ctrl+drag the circle to create its copies for San Francisco, New York, Orlando, Houston (or any other cities you like).
5. Select all circles (use dragging the selection rectangle), but do not include the map in this selection. If the map gets selected Shift+click it to deselect.
6. Right-click one of the circles and choose **Grouping | Create a group** from the context menu. All circles should now be in one group.
7. Open the **Controls** palette and drag the **Radio buttons** object. Drop it on the left of the map.
8. In the **General** properties of the radio buttons specify the following list of choices: All, Only map, Only cities.
9. Select the map and set its **Visible** property on the **Dynamic** page to: **radio.getValue() != 2**.

10. Select the group of cities (by clicking one of the circles) and set its **Visible** property on the **Dynamic** page to: **radio.getValue() != 1.**
11. Run the model. Switch radio buttons.

In this example the radio buttons are not linked to any variable, but their API, namely the method **getValue()**, is used in the dynamic properties of shapes controlling the shape visibility.

The value of the radio buttons controls is integer, not string, and it starts at 0.

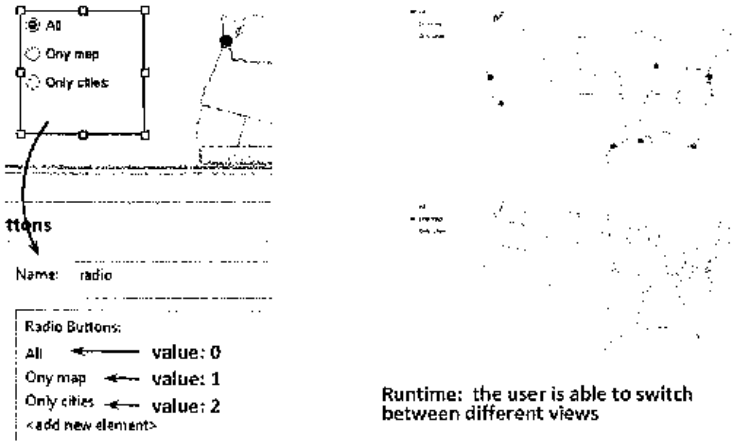


Figure 13.5 Radio buttons used to control the visibility of the map and the cities

Example 13.5: Combo box controlling the simulation speed

We will use combo box to control the speed of simulation. Although there is a toolbar section **Time scale setup**, such control may be useful to provide a limited choice of speeds making sense for a particular model.

▶ **Follow these steps:**

1. Drag the **Combo box** from the **Controls** palette to the canvas and extend it as shown in Figure 13.6.
2. In the **General** page of the combo box properties enter the following items in the **Items** list:
   x1 (real time)
   x10
   As fast as possible
3. Enter this code in the Action field of the same property page:

```
if( value.equals( "As fast as possible" ) )
```

```
  getEngine().setRealTimeMode( false );
else {
  getEngine().setRealTimeMode( true );
  if( value.equals( "x1 (real time)" ) )
    getEngine().setRealTimeScale( 1 );
  else if( value.equals( "x10" ) )
    getEngine().setRealTimeScale( 10 );
}
```

4. To view the simulation speed change, create a simplest process model consisting of a **Source** object connected to a **Sink** object (drag both objects from the **Enterprise Library** palette and connect their ports).
5. In the **Projects** tree select the **Simulation** experiment of the model. In the **Model Time** page of the experiment properties set **Stop** to **Never**.
6. Run the model. Set different speeds using the combo box. Observe the speed at which the entities are generated.

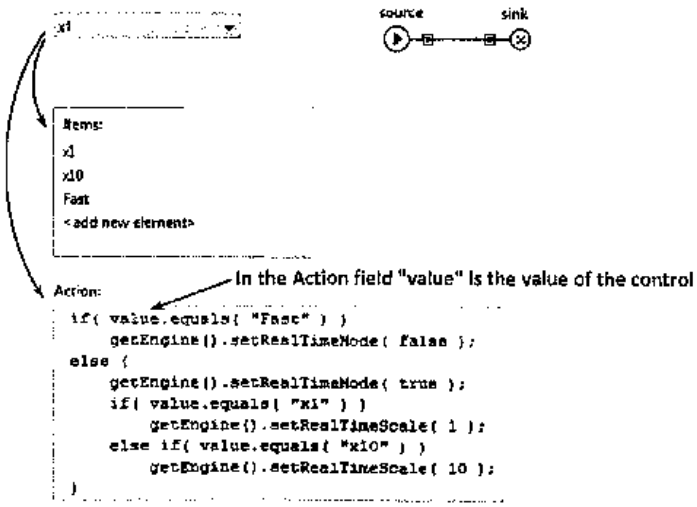


Figure 13.6 Combo box - custom control of the simulation speed

The combo box value is a string, therefore in the Action field we need to compare the value to the three different **String** constants. The reason why strings are compared using **equals()** method and not == operator is explained in Chapter 10, "Java for AnyLogic users".

Example 13.6: File chooser for text files

In this example the **File chooser** will be combined with the **Text file** object so that the user of the model will be able to choose a file (e.g. with the model parameters) before starting the model.

▶ **Follow these steps:**

1. Drag the **File chooser** from the **Controls** palette to the canvas and extend it a bit as shown in Figure 13.7.
2. Open the **Connectivity** palette and drag the **Text File** object (see Section 11.1) to the right of the file chooser.
3. Open the **Presentation** palette and drag the **Text** object below the file chooser, as shown. Set the font size of the text to 11pt.
4. In the **General** page of the file chooser properties set the **Title** field to **Open a text file**
5. In same property page add a file filter item:
   File name: Text files
   File extension: txt
6. In the Action field of the file chooser properties write this code:

```
file.setFile( value, TextFile.READ );
text.setText( "" );
while( file.canReadMore() )
  text.setText( text.getText() + file.readString() + "\n" );
```

7. Run the model.
8. Click the file chooser button. Select any text file and press **Open**. The file contents will appear on the screen.

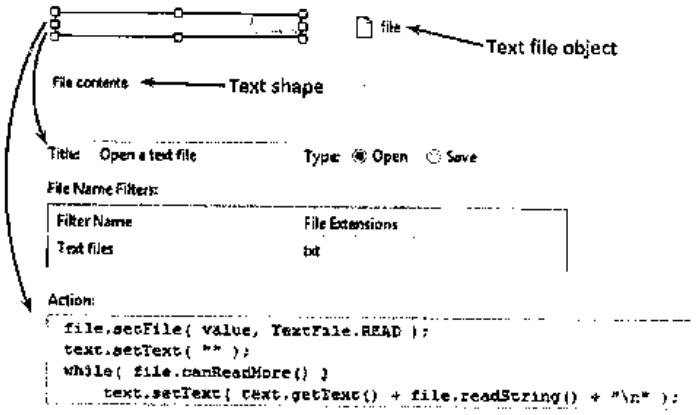


Figure 13.7 File chooser works with the Text file object

The file chooser is set up as follows. The title field becomes the title of the **Open file** dialog. The name filter tells the dialog to show only files with extension "txt". When the user chooses a file, the file chooser action gets executed. It sets up the chosen file to the Text file object **file** – the object that can read and write text files. Then the text of the **text** shape is cleared (we assign an empty string "") and then the whole contents of the text file is added to the text shape line by line ("\n" is the end of line symbol).

Indivisibility of control actions and model events

The actions associated with controls are initiated by the user and may be executed during model runtime. Therefore you may wonder how they are synchronized with the events in the model.

AnyLogic guarantees indivisibility of all control actions and all model events. It means that the control action code is executed without discontinuities (is atomic) and is never interrupted by the execution of the model events and vice versa. If the user changes the control state during the continuous phase of the model execution ,such as when the dynamic equations are being solved numerically, the control action is treated as yet another discrete event, so the solver correctly stops before the control action and resumes after.

13.1. Dynamic properties of controls

Just as is done for shapes, the dynamic properties of controls are constantly evaluated at runtime and may be used to dynamically change control availability, visibility, size, position, etc. The dynamic properties are largely located on the **Dynamic** property page. Some frequently used ones are located on the **General** page.

Almost all controls have the **Enabled** property. If you enter a boolean expression there, the control will be enabled if it evaluates to **true**, and disabled otherwise (see Example 13.2: "Buttons changing the parameter value"). The **Visible** property located on the **Dynamic** page is used to temporarily hide controls. The size properties are used rarely as typically you do not want to resize the controls at runtime.

Example 13.7: Radio buttons enabling/disabling other controls

You can create sophisticated "dialog-like" behaviors by binding the **Visible** and/or **Enabled** properties of some controls to the states of other controls. Suppose you want to suggest two modes to the user: the use of default parameters or a custom setup. You can use radio buttons to enable/disable controls linked to the model parameters.

▶ **Follow these steps:**

1. Create a group of two radio buttons by dragging the **Radio buttons** object from the Controls palette.
2. In the properties of the radio buttons set the first button label to "Use default settings" and the second – to "Use custom settings".
3. Create a slider below the radio buttons as shown in Figure 13.8. You may need to adjust the radio buttons size to avoid overlapping with the slider.
4. Create a parameter on the right-hand side of the slider. Set the default value of the parameter to **50**.
5. In the **General** page of the slider properties set:
   Link to: checked, value: **parameter**
   Enabled: **radio.getValue() == 1**
6. In the **General** page of the radio button properties write the following code in the **Action** field:

```
if( value == 0 )
  set_parameter( 50 );
else
  set_parameter( slider.getValue() );
```

7. Run the model. Play with radio buttons and the slider. Note how the parameter value changes

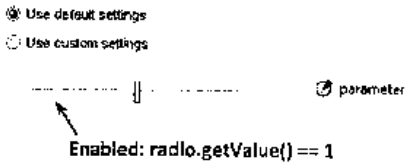


Figure 13.8 Radio buttons enable and disable the slider

Note that the slider remembers its position while the radio buttons are switched to the first option. Therefore when the user switches to the second option, we force the parameter value to the slider value.

Example 13.8: Keeping controls in the top left corner of the window

Suppose you have a control or a group of controls that you wish to appear in the top left corner of the model window regardless of how the user pans the canvas or which view area he chooses. In order to achieve this we can use the dynamic **X** and **Y** properties of the control (or of the group – remember that controls can be grouped in the same way as shapes).

▶ **Follow these steps:**

1. Create a button and place it on the coordinates (50,50).

2. Create a circle anywhere nearby – it will just show how we pan the canvas.
3. In the Dynamic page of the button properties set:
   X: -getPresentation().getPanel().getOffsetX() + 50
   Y: -getPresentation().getPanel().getOffsetY() + 50
4. Run the model. Drag the canvas by moving the mouse with the right button pressed. The circle will move. The button will not.

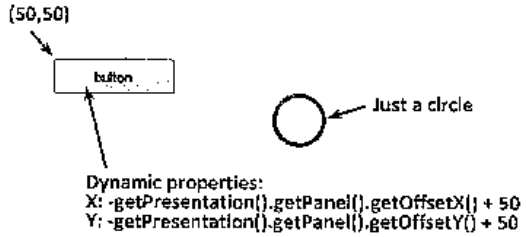


Figure 13.9 This button will always stay at (50,50) relative to the model window

The expression `getPresentation().getPanel().getOffsetX()` returns the X-offset of the presentation coordinate origin (0,0) from the top left corner of the model window. Therefore, to keep the button at X=50 we need to shift it in the opposite direction by the same distance, and add 50, which is the original (static) button offset. The same is true for Y. You may extend this example by adding multiple view areas and switching between them.

Example 13.9: Replicated button

Sometimes you need to create an array of similar controls e.g. to be able to change an array of similar objects at runtime. AnyLogic *replicated controls* may save you some drawing time and also may make your model scalable. In this example we will create a replicated button and use it to change the fill color of a replicated shape (see Section 12.4). Moreover, the number of copies of the button and the shape will be dynamically controlled by a slider.

▶ **Follow these steps:**

1. Create the following four objects as shown in Figure 13.10: a slider, a variable, a button and a rounded rectangle.
2. Set the name of the variable to **N**, its type to **int**, and initial value to **3**.
3. Link the slider to the variable **N** and set its minimum and maximum values to **1** and **9** correspondingly.
4. Set the following dynamic properties of the rounded rectangle:
   Replication: N
   Y: 100 + 50 * index

5. Set the following dynamic properties of the button:
   Replication: N
   Y: 100 + 50 * index
   Label: "Paint shape " + index
6. On the General page of the button properties set the Action to:
   roundRectangle.get(index).setFillColor(blueViolet);
7. Run the model. Move the slider and press the buttons.

As you can see, the number of buttons changes as the slider changes the variable N. By using the index of the button copy in the button Action field you can do different things with different buttons, in this case - change the color of the shape whose number equals the number of the button. The label of the button also depends on the index.

If you paint the shape with a number, let's say 8, then set the number of shapes to 5 and back to 9, the new 8th shape will be not painted because it is a brand new object. The old one has been completely deleted.

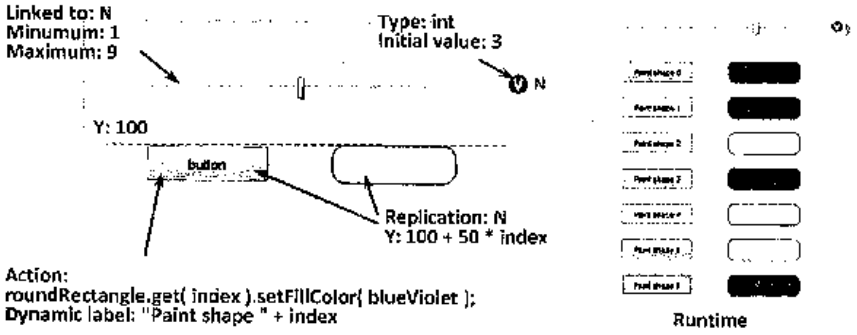


Figure 13.10 Replicated button controlling a replicated shape

13.2. Controls' API

Just like anything in AnyLogic, the controls are mapped to Java objects and expose their API (application program interface) to the modeler. In many cases the same effect can be achieved by using either the dynamic properties of a control or calling its methods. The remark in Section 12.5 will help you to make your choice.

The most frequently used methods of controls are:

- **setVisible(boolean)** – hides or shows the control
- **setEnabled(boolean)** – enables or disables the control
- **action()** – executes the action of the control

- setValueToDefault() – sets the value or the control to the default one
- setValue(..., boolean) – sets value to a given one and optionally calls the action

The full list of methods is available in *AnyLogic Classes and Functions: API Reference* (The AnyLogic Company, 2013). In Figure 13.11 below we give the Java class hierarchy for controls. Note that the base class for all controls is **ShapeControl**, which is a subclass of **Shape**, therefore controls implement the methods of **Shape**. The class **ShapeTextField** corresponds to the edit box control.

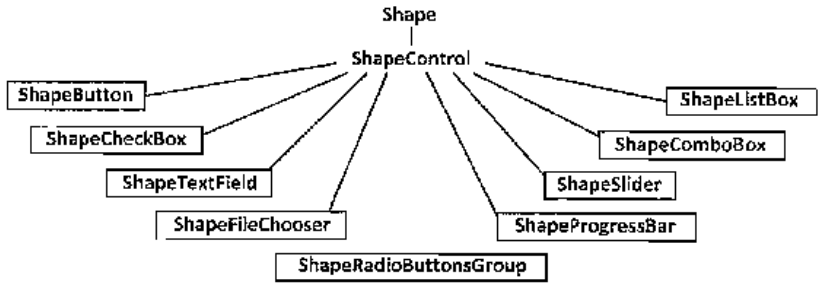


Figure 13.11 AnyLogic Java classes for controls

13.3. Handling mouse clicks

Handling mouse clicks is yet another way to add interactivity to your model. You can use click handling to display additional information on the objects (see Example 12.10: "Show/hide a callout"), to create hyperlinks, to define locations on the map, to control specific elements of the model, and so on.

Mouse clicks in the model window are processed as follows.. AnyLogic iterates through all shapes in their Z-order (see Section 12.1), starting from top. If a shape area contains the coordinates of the click and the shape's **On click** action is defined, the action is executed (and, by default, returns **false**). If the action returns **true** (which you need to do explicitly), the click processing will stop. Otherwise the iteration continues down to the last shape at the bottom, see Figure 13.12.

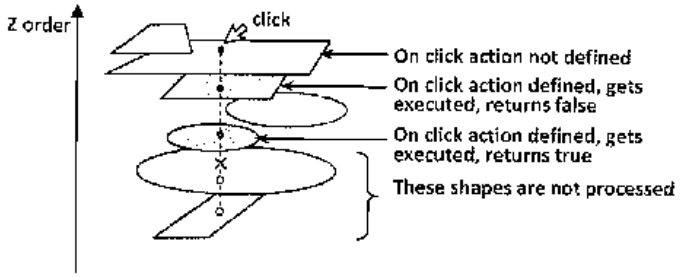

Z order
click
On click action not defined
On click action defined, gets executed, returns false
On click action defined, gets executed, returns true
These shapes are not processed

Figure 13.12 AnyLogic processing of mouse clicks

In the **On click** code field you can also access the exact coordinates of the click relative to the shape coordinates. They are available as **clickx** and **clicky**.

Example 13.10: Hyper link menu to navigate between view areas

Click handling is often used to provide custom navigation in the model. Modelers create text or graphics and make them hyper links to various important locations. In this example we will show how to create a simple hyperlink menu to switch between two locations marked with view areas. An area may contain the model animation while another might contain the model output.

▶ **Follow these steps:**

1. Create a new model. Find the default view area **origin** at the coordinate origin of the **Main** active object.
2. In the **General** property page of the view area at (0,0) change:
   Name: viewAnimation
   Title: Animation
3. Create another view area at (0,600). The **View area** object is located in the Presentation palette.
4. In the **General** property page of the second view area set:
   Name: viewOutput
   Title: Output
5. Draw a circle or any other graphics in the center of the **Animation** view, i.e. at (400, 300) just to identify the view at runtime.
6. Drag a time chart or any other chart from the **Analysis** palette in the center of the **Output** view, i.e. at (400, 900) for the same purpose.
7. Create two texts (text shapes) "Animation" at (50,20) and "Output" at (150,20). Set their font size to 16, Bold.
8. Set the color of the text "Output" to **blue**.
9. In the **Dynamic** property page of the text "Output" write the following code in the **On click** field: **viewOutput.navigateTo();**

10. Draw a blue line under the text "Output" to make it look like a hyperlink, see Figure 13.13.
11. Select both texts "Animation" and "Output", and the blue line. Ctrl+drag the selection to create a copy.
12. Drag the copy downwards to the second view area until the text "Animation" is at (50, 620).
13. Set the color of the second "Animation" text to blue, and "Output" – to black.
14. Move the blue line from "Output" to "Animation" and extend it to match the text.
15. Cut the code from the **On click** field of the **Dynamic** property page of the second "Output" text to the same field of the "Animation" text and change it to: **viewAnimation.navigateTo();**
16. Run the model. Click the hyperlinks you created.

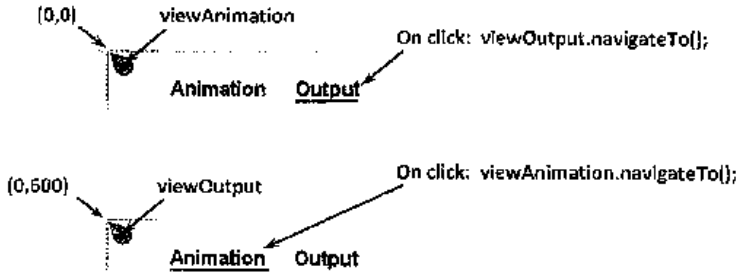


Figure 13.13 Hyperlink navigation between view areas

The blue texts in this example are click-sensitive. Their **On click** actions call the method **navigateTo()** of the two view areas, which displays the corresponding part of the canvas.

If you do not care how the click is further processed (as in the case where there are no shapes below the click-sensitive ones, or those shapes are click-insensitive), you can omit the **return** statement at the end of the **On click** code – just as we did it in this example.

Example 13.11: Creating dots at the click coordinates

You can not only find out that a certain shape was clicked, but also the exact coordinates of the click (relative to the shape). In this example we will use that to create small dots at the locations of the clicks.

▶ **Follow these steps:**

1. Create a rectangle with the upper left corner at approximately (50,50) and size 500 x 500.

2. Give the rectangle the name **clickArea**.
3. In the **Dynamic** page of the rectangle properties write this code in the **On click** field:

```
ShapeOval dot = new ShapeOval();
dot.setRadius( 2 );
dot.setFillColor( blue );
dot.setLineColor( null );
dot.setPos( clickArea.getX() + clickx, clickArea.getY() + clicky );
presentation.add( dot );
```

4. Run the model. Click within the rectangle bounds.

The **clickx** and **clicky** variables available in the **On click** field are the coordinates of the click *relative to the shape that catches the click*. To transform them to the absolute coordinates we need to add the coordinates of the shape itself (assuming the shape belongs to the top level **presentation** group.

Example 13.12: Catching mouse clicks anywhere on the canvas

You already know that mouse clicks are handled by shapes. What if you need to catch mouse clicks anywhere on the canvas? One of the ways to do it is to programmatically create a very large invisible shape that handles the clicks.

▶ **Follow these steps:**

1. Create a small circle (radius 5 pixels) anywhere on the canvas. Leave its default name **oval**. We will use that circle to show the click location.
2. In the **Advanced** properties page of the active object write the following code in the **Additional class code** field:

```
class ClickDetector extends ShapeRectangle {

  ClickDetector() {
    super( true, -100000, -100000, 0, null, null,
        200000, 200000, 0, LINE_STYLE_SOLID );
  }

  @Override
  public boolean onClick( double clickx, double clicky ) {
    clickx += getX();
    clicky += getY();
    oval.setPos( clickx, clicky ); //The click handling code
    return false;
  }

}
```

3. In the **General** page of the active object class properties write in the **Startup code** field: `presentation.add( new ClickDetector() );`

4. Run the model. Click in different places.

Since we can create shapes dynamically in AnyLogic, nothing prevents us from creating a very large shape that will cover all the meaningful area of the model presentation. To make that shape click-sensitive we create a custom subclass of the **ShapeRectangle** and override its method **onClick()**, where we define the reaction – in this example it is just by placing the circle at the click location. In the startup code we create an instance of the click-sensitive rectangle and add it to the top-level **presentation** group.

Chapter 14. 3D animation

AnyLogic supports both 2D and 3D space in simulation models, and enables you to create high-quality interactive *3D animations* in addition to more technical-looking 2D animations. You can define a 3D scene, use the standard shapes provided in the 3D palette, imported 3D graphics, or include 3D objects composed of primitive shapes you create yourself. You can associate the 3D objects with entities, pedestrians, rail cars, and vehicles. Agents can live and move in 3D space. You can view 3D animation in one or multiple 3D windows simultaneously with 2D animation. 3D animation works everywhere: when running the model from within the AnyLogic development environment, exported as a Java application, or published on the web as a Java applet.

AnyLogic graphical editor is two-dimensional, and the natural scenario of building a 3D animation starts with a 2D (X,Y) plane, upon which you draw the "XY projection" of the scene, and then "grow" the picture into the third, *Z-dimension* (see Figure 14.1). The models originally designed as 2D can be converted into 3D easily by defining the Z-properties of the 2D shapes.

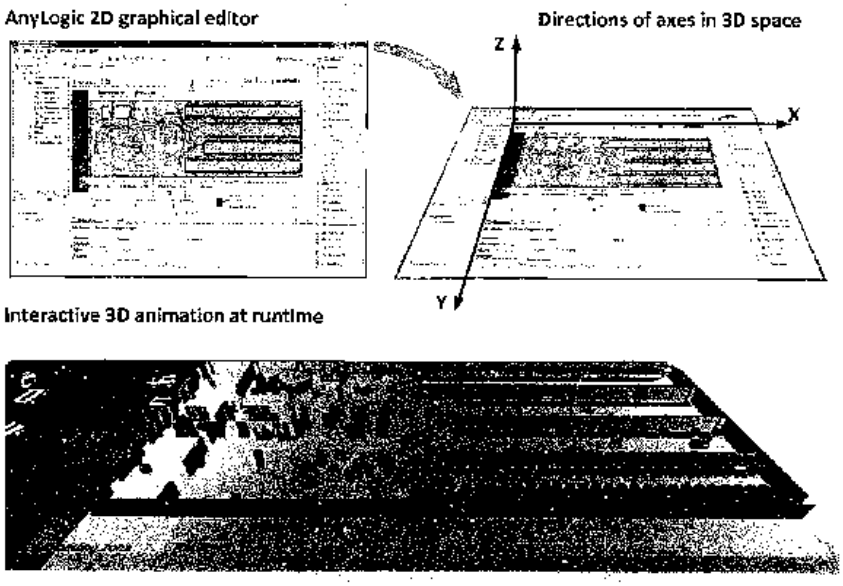


Figure 14.1 Creating a 3D animation

There are two palettes for 3D animation: **3D** and **3D Objects** (see Figure 14.2). The first one contains primitive shapes and 3D-specific objects, such as **3D window** or

Camera. The second palette contains frequently used 3D graphics such as a person, car, forklift truck, house, etc., that can be associated with static or dynamic objects in the model.

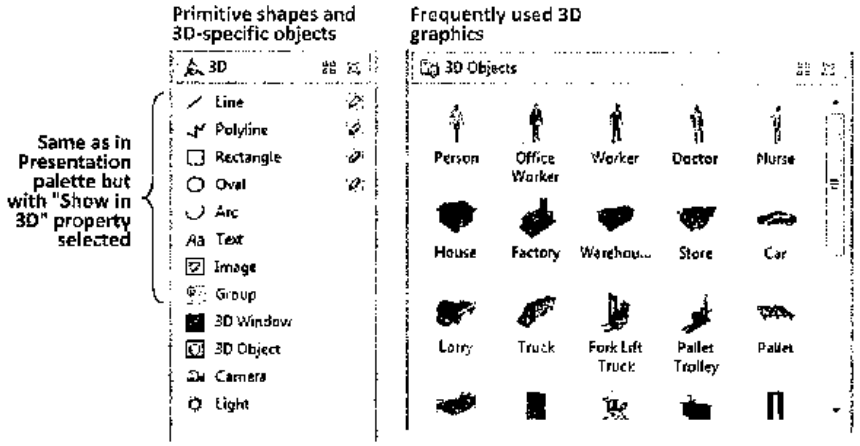


Figure 14.2 3D-related palettes

Example 14.1: A very simple model with 3D animation

We will create a very simple model with 3D animation. In this model, people will exit a house and walk to a store located nearby.

▶ **Follow these steps:**

1. Create a new model.
2. Put together a simple flowchart as shown in Figure 14.3. Select **Maximum capacity** for the **delay** object, and leave all other parameters as is.
3. Open the **3D** palette and use it to draw a background rectangle and a polyline as shown in Figure 14.3. The shapes dragged from that palette differ from the same shapes from the **Presentation** palette only in that they have the property **Show in 3D scene** selected by default.
4. On the **Advanced** property page, set **z** and **z-height** of the shapes, as shown.
5. Open the **3D Objects** palette and drag the **House** and the **Store** objects on the background rectangle.
6. From the same palette, drag the **Office Worker** anywhere on the canvas. It looks like a bug, but don't worry; this is the top view of the office worker.
7. Set the **Entity animation shape** of the **source** object to **officeWorker** and set the **Animation guide shape** of the **delay** object to **polyline**.
8. Drag the **3D window** from the 3D palette and place it below the flowchart.
9. Run the model.

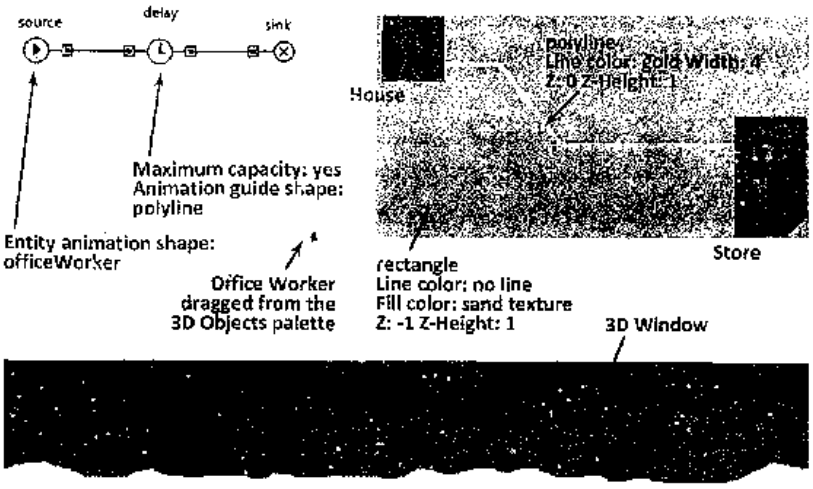


Figure 14.3 The "source code" of a simple model with 3D animation

At runtime, you are able to view 2D and 3D animations simultaneously. In the 3D window you can drag the scene, use the mouse wheel to zoom, or use Alt+drag to rotate the view.

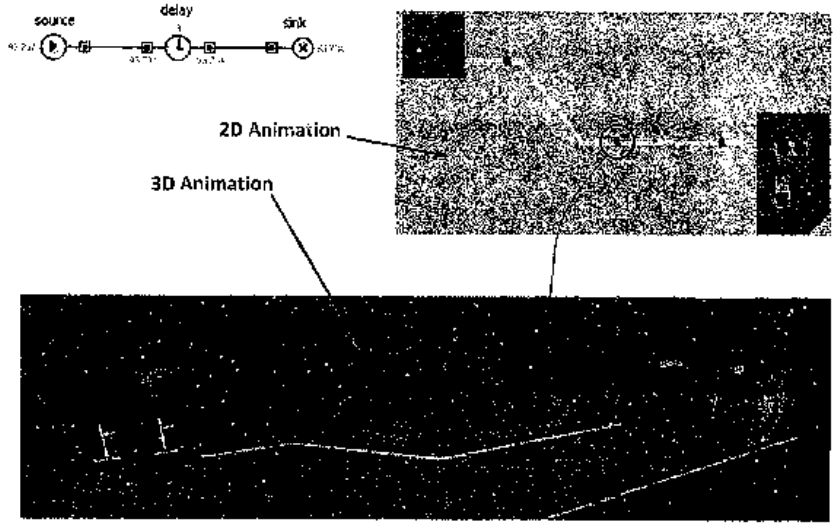


Figure 14.4 "A very simple 3D model" at runtime

14.1. Primitive 3D shapes

You may have noticed that the primitive shapes in the 3D palette are the same as in the **Presentation** palette. Indeed, they are in fact the same objects, with the only difference being that their **Show in 3D scene** property is selected by default. Therefore, by default, they will appear in the 3D scene.

In 3D, the rectangle shape will become parallelepiped (a solid whose faces are all parallelograms) and the oval will become a cylinder (see Figure 14.5). The Z-coordinate of their bottoms, along with their Z-heights, are defined on the **Advanced** properties page. Do not confuse the **Height** of the rectangle and its **Z-Height**: the former actually becomes "Y-Height" in 3D.

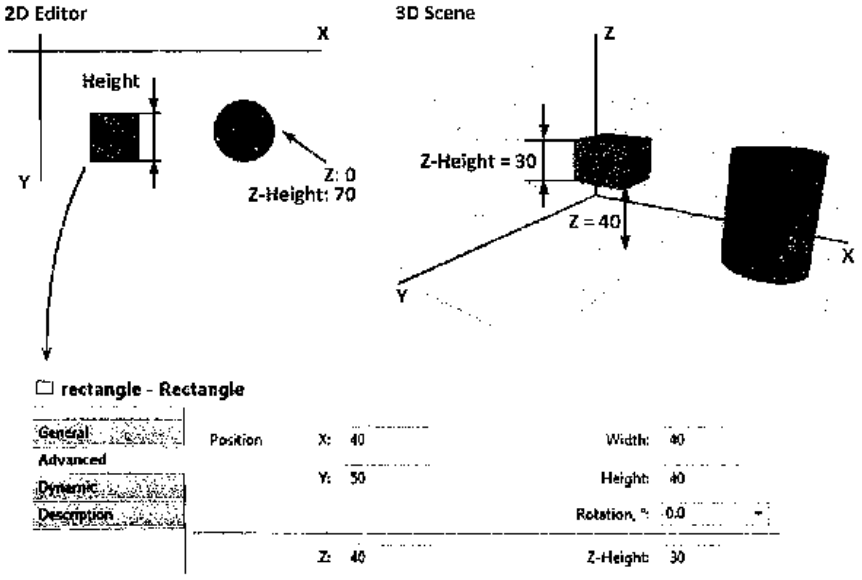


Figure 14.5 3D Rectangle and 3D Oval. Z and Z-Height properties

Lines, polylines, and arcs have additional 3D properties. Lines and arcs have **dz** – the difference in the Z coordinate of their start and end points. For a polyline, you can specify the Z-coordinate of each point in the **Points** property page. The 3D versions of these shapes are shown in Figure 14.6.

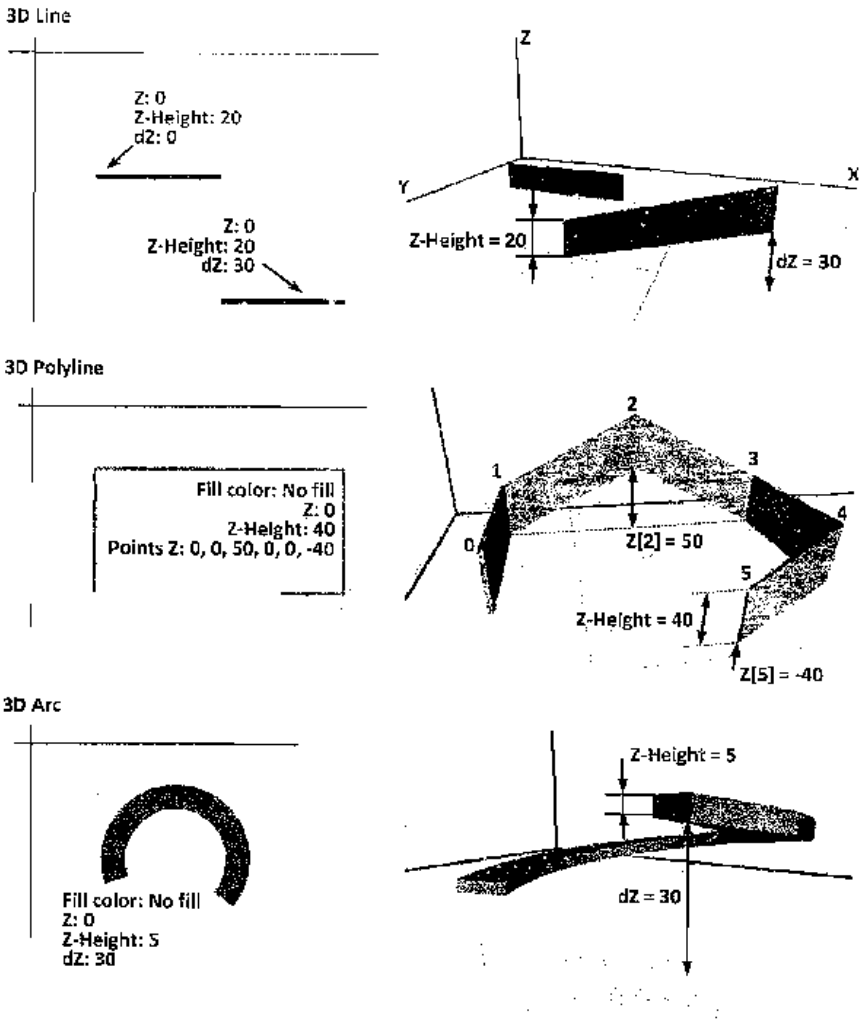


Figure 14.6 3D Line, 3D Polyline, and 3D Arc. dZ and Point Z properties

If **Line color** is set for a 3D shape, it becomes the color of its "walls," both outside and inside. **Fill color** becomes the color of its "bodies" (see Figure 14.7).

Please note that if any two points of a polyline or start and end point of an arc have different Z coordinates, these shapes *cannot be filled in 3D*. They can only be filled if they have "flat surface."

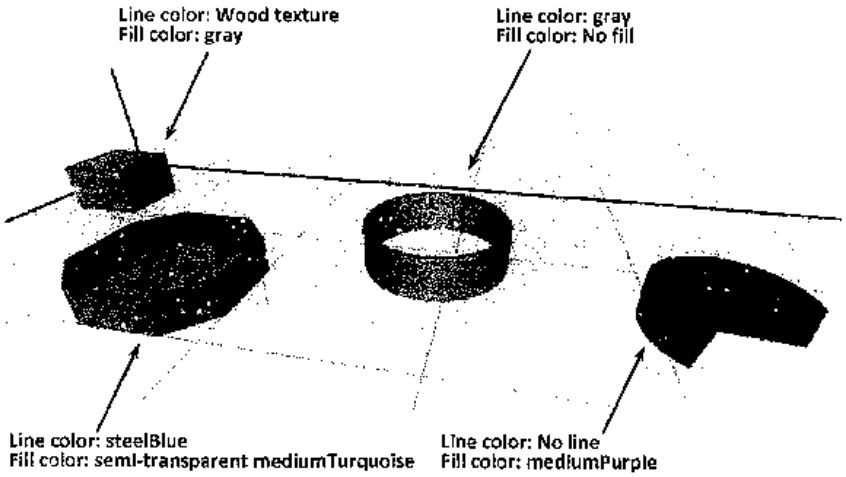


Figure 14.7 Line and fill colors of 3D shapes

Images and text can also appear in a 3D scene. Both have no Z-height (they are infinitely thin) and can be viewed both from the top and from underneath.

If two or more shape surfaces happen to be in the same plane and they intersect, the 3D renderer will not know how to display them. Undesired visual effects may then occur. To avoid this, you should slightly shift one of the surfaces by making the corresponding coordinates a bit different.

Consider Figure 14.8. To make the text appear nicely on top of the map, its Z coordinate was set to 1. Sometimes, one unit of difference is too much; in that case, you can set it to a fraction, say, 0.2 (this can be done in the dynamic properties of the shape).

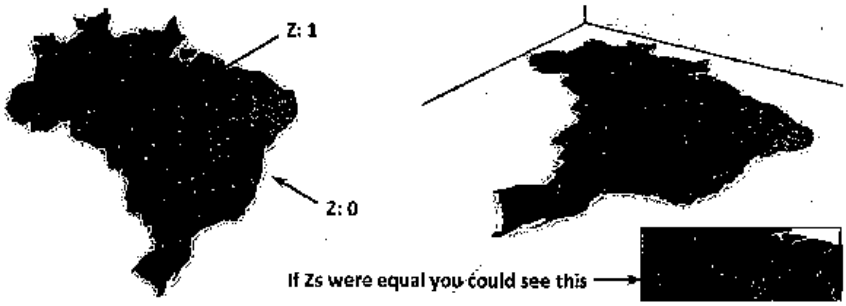


Figure 14.8 Image and text in 3D scene. Undesirable rendering effects

The 2D order of shapes (the order that is controlled by the Send to Back/Bring to Front commands) does not affect the 3D scene at all. The shape that appears underneath another shape in 2D can be above it in 3D!

The 3D-related properties of shapes can be dynamic (see Section 12.3) and can be linked to model variables and expressions, just like all other properties (see the **Dynamic** property page of a 3D shape). The **Rotation** of a shape is called **Rotation Z**, if the shape is marked as 3D. This is because the 2D rotation is the rotation in the (X,Y) plane around the Z axis.

You will find no fields for rotation in the (Y,Z) plane ("Rotation X") and rotation in the (X,Z) plane ("Rotation Y"). These cannot be controlled for individual shapes, but can be controlled at the 3D group level.

14.2. 3D groups and rotation

3D shapes can be grouped in the normal way. The group's **Show in 3D scene** property "owns" the same property of all the group members. If a shape being grouped is in 3D, the group is also marked as 3D, and so are all other shapes in the group. Therefore, either all shapes in a group are shown in a 3D scene, or none of them are.

3D groups have dynamic **Rotation X** and **Rotation Y** properties, which allows you to rotate 3D shapes in the (Y,Z) and (X,Z) planes – this is the *only* place where you can do it.

Example 14.2: Rotation in 3D – a sign on two posts

We will draw a sign with two posts. At design time, the sign will lie on the ground, i.e. in the (X,Y) plane. However, at runtime, it will be raised vertically by using the 3D group rotation properties.

▶ **Follow these steps:**

1. Create a new model.
2. Draw the picture (as shown in Figure 14.9) using the text, rectangle, and line objects from the **3D** palette. Set the Z-properties of the shapes as shown.
3. Select all shapes and group them. A new group is created, with the center approximately at the center of the picture.
4. Click the group and choose **Select group contents** from the context menu.
5. Move the group members up so that the group center is at the bottom of the posts (see Figure 14.9 on the right).

6. Select the group and open its Dynamic properties page. Set **Rotation X** to **-PI/2**.

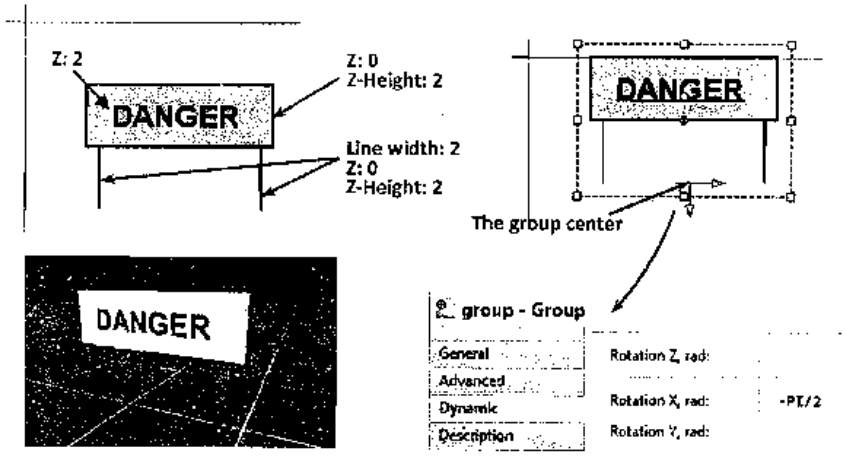


Figure 14.9 A 3D group rotated in the (Y,Z) plane

7. Drag a **3D window** anywhere on the canvas. In its **Scene** properties page, set **Grid color** to **white**.
8. Run the model. The sign is displayed as being raised up, but the text appears distorted because its surface intersects with the surface of the sign (text's Z = 2 and it has no Z-height, and the sign's Z + Z-Height = 2 as well).
9. To fix this, open the **Dynamic** page of the text properties and set **z** to **2.1**. This will position the text slightly above the sign surface.
10. Run the model again.

Example 14.3: Bridge crane 3D

We will build a 3D model of a bridge crane. The bridge crane has three degrees of freedom: it moves along the rails (in the X dimension), its hoist moves along the bridge (the Y dimension), and the load moves up and down (the Z dimension). The crane will be controlled by three sliders. By moving a slider, you can define the target coordinate for X, Y, or Z, and the crane will start moving. We will use a number of 3D shapes along with a 3D group with dynamic properties.

▶ **Follow these steps:**

1. Create a new model. Open the **Window** page of the **Simulation** experiment properties, and set the dimensions of the window to 900 x 700.
2. Open the editor of the **Main** object. Draw the crane "XY projection" with three 3D rectangles and two 3D circles, as shown in Figure 14.10.

3. Group all these shapes. Move the group contents, and then the group, so that the group center is at the coordinate origin, as shown in Figure 14.10 on the right.

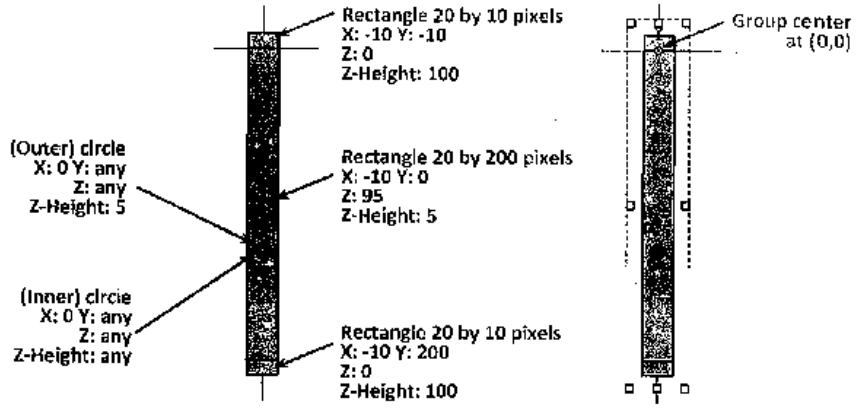


Figure 14.10 The (XY) projection of the bridge crane

4. Drag the **3D Window** from the **3D** palette. Move its top left corner to (0,400) and resize the window to 900 x 300 pixels.
5. Run the model and view the (static at this point) bridge crane.

▶ **Create the model of the crane movement in X dimension**

6. Add a parameter **speed** with a default value of **50**. For simplicity, we will use the same speed for movement in all three dimensions.
7. Add an event **moveX**, two variables, **startX** and **targetX**, and a function **X()**, as shown in Figure 14.11.

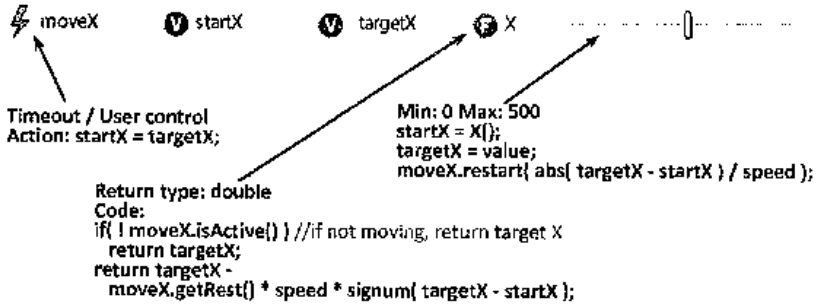


Figure 14.11 The model of crane movement in X dimension

The function **X()** returns the current X coordinate of the crane. If the crane is not moving (meaning the event **moveX** is inactive), the current X equals the target X. Otherwise, we calculate the distance to cover, which is the product of speed and time to target, and subtract it from the target X. The slider sets the start X to the current X and starts a new movement by scheduling **moveX** at the time of arrival.

8. Select the group and define its dynamic X as **X()**, so that the X coordinate of the crane animation always equals the current X of the crane.
9. Run the model and move the slider. The crane moves along the X axis. If you move the slider while the crane is in motion, it cancels the previous command and moves to the new target location.

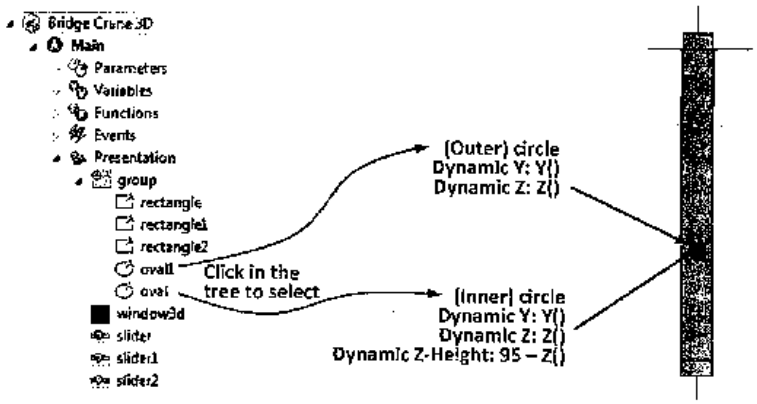


Figure 14.12 Dynamic properties of the hoist

Add movement in the Y and Z dimensions

10. Create two copies of the five objects (event **moveX**, variables **startX** and **targetX**, function **X()**, and **slider**).
11. Rename the objects in the first copy by replacing "X1" with "Y", and in the second copy by replacing "X2" with "Z", so that you have **moveY**, **moveZ**, etc.
12. Edit the code of events **moveY** and **moveZ**, functions **Y()** and **Z()**, and the two sliders, respectively, by replacing "X" with "Y" and "Z".
13. Set the range of the slider for Y to 0-200, and the range for Z 0-95.
14. Specify the dynamic properties of the circles that represent the hoist and the load, as shown in Figure 14.12. Use the **Projects** tree to select small shapes.
15. Run the model and play with the crane.

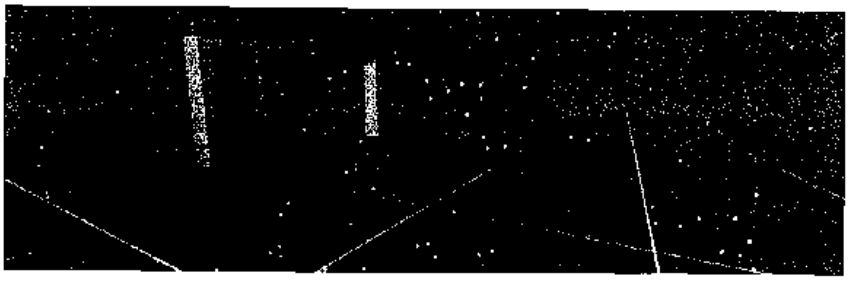


Figure 14.13 3D animation of the bridge crane

14.3. Standard and imported 3D graphics

Primitive 3D shapes are primarily used to create simple *3D objects*, or objects that dynamically change color, size, or have moving parts, like the bridge crane in the previous section. To create more realistic (and attractive) 3D animations, you can use sophisticated 3D graphics (also called 3D models), both standard included in AnyLogic, as well as your own.

Using standard 3D graphics

A collection of pre-drawn 3D graphics are available for your use in the **3D Objects** palette (see Figure 14.14).

▶ **To use a standard 3D graphics:**

1. Drag the 3D object from the 3D Objects palette on the canvas.
2. Choose the scale and orientation (the **Axis order** property)

By default, the objects in the palette are put on the (XY) ground plane, and are therefore seen in the graphical editor from the top. However, you can change this (see Figure 14.14). The default scale of objects is chosen to match the default library settings. For example: people, cars and trucks, trolleys, and security control devices are all in the same scale, which is consistent with the default scale of the Pedestrian Library. The scale of the rail cars and the containers matches the default scale of the Rail Library (see Chapter 9).

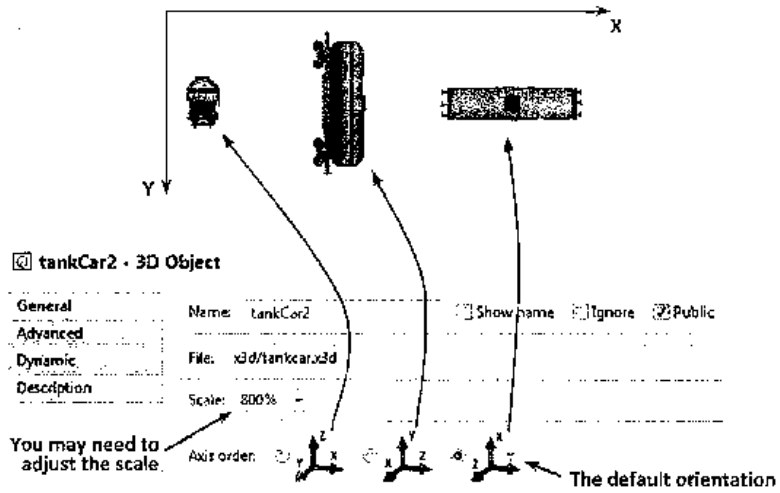


Figure 14.14 Orientation of 3D objects

Using external 3D graphics

AnyLogic uses *X3D file format* for 3D objects. X3D is the ISO standard XML-based file format for representing 3D computer graphics, and is the successor to the Virtual Reality Modeling Language (VRML), ("X3D," n.d.). A good collection of X3D resources can be found at (Web3D Consortium, 2013).

While X3D does not have as many 3D models in public access as, say, Google Sketchup™, there are free and commercial conversion tools: Vivaty Studio™, Blender™, and Google Sketchup Pro™, for example.

▶ **To import Google Sketchup graphics into AnyLogic using Vivaty Studio:**

1. Download Vivaty Studio from (Web3D Consortium, 1999-2013) and install it. This is free of charge.
2. Go to 3D Warehouse (Trimble Navigation Limited & Google, 2013) and choose a 3D model to download. The model should be downloadable in either the *Google Earth* (.KML and .KMZ) or *Collada* (.DAE) formats; Vivaty Studio cannot read the .SKP file directly.
3. Save the Sketchup 3D model. If it is in Collada format, it is usually a ZIP archive, which you will then need to unzip.
4. Run Vivaty Studio and create a new project.
5. Choose **File | Import Collada** or **Import Google Earth**, depending on what you have downloaded. Import the model.
6. Choose **File | Save** to save the model (this is required before you can export it in X3D).

7. Choose **File | Export X3D or VRML**. In the **Save As** dialog select the **Uncompressed** option. Save the X3D file.
8. In AnyLogic, open the **3D** palette and drag the **3D Object** on the canvas.
9. In the 3D object properties, choose the X3D file saved by the Vivaty Studio. Choose the appropriate scale and orientation.

Similarly, you can import the Sketchup graphics using the (Blender Foundation, 2013).

14.4. Hierarchical 3D animations. Embedded 3D presentation

Just like with 2D animation, 3D animation in AnyLogic is hierarchical. The 3D presentation of an embedded active object can appear in the 3D scene of its container object. The embedded object presentation has the **Show in 3D Scene** and the **Z** properties.

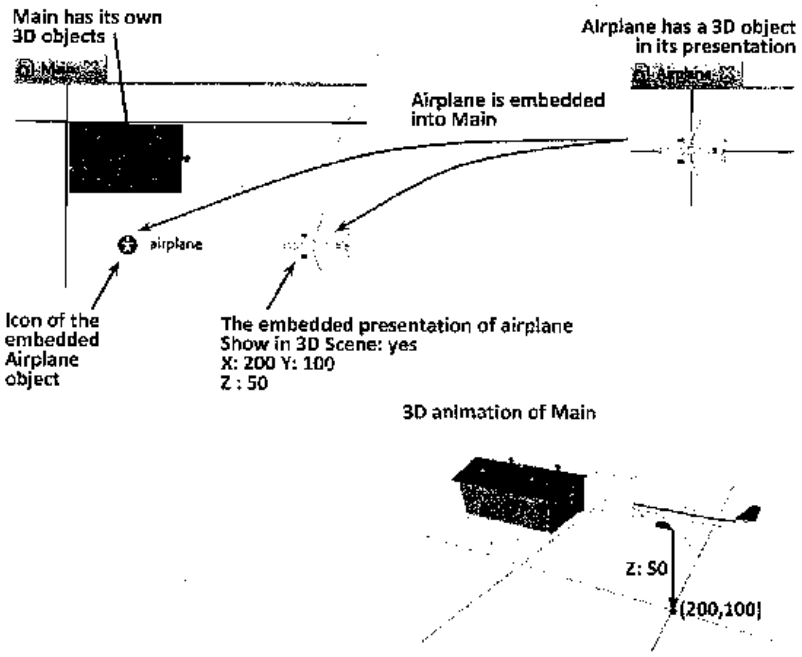

Main has its own 3D objects
Airplane has a 3D object in its presentation
Airplane is embedded into Main
airplane
Icon of the embedded Airplane object
The embedded presentation of airplane
Show in 3D Scene: yes
X: 200 Y: 100
Z : 50
3D animation of Main
Z: 50
(200,100)

Figure 14.15 Embedded 3D presentation

The point of origin of the embedded airplane presentation is placed into its (X,Y,Z). Therefore, if you ask the airplane to move to (-100, 0, 100), it will arrive at (100, 100, 150) on Main.

The airplane active object has its own 3D scene (in our case, it only contains the airplane picture). This can be viewed by adding a 3D window in the Airplane editor and navigating to the airplane object at runtime.

14.5. 3D Windows

To view 3D animation in AnyLogic, you use *3D windows*. The **3D Window** object is located in the **3D** palette. In the **Scene** properties page of the 3D window, you can set up the background color and (optionally) the grid color. These properties are shared by all 3D windows in the active object class. The grid is drawn on the (X,Y) plane (see Figure 14.16) and has a step of 100 pixels.

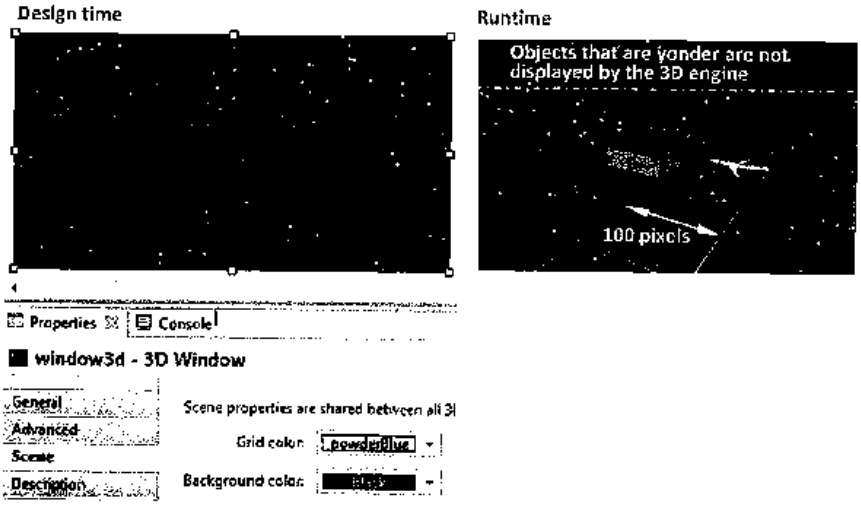


Figure 14.16 3D window

Navigation in the 3D scene at runtime

If you do not associate the 3D window with a camera (see Section 14.6), in the beginning it will take the default view on the 3D scene, trying to display the entire scene from some perspective. You can change the viewpoint by using the mouse and the Alt key (see Figure 14.17).

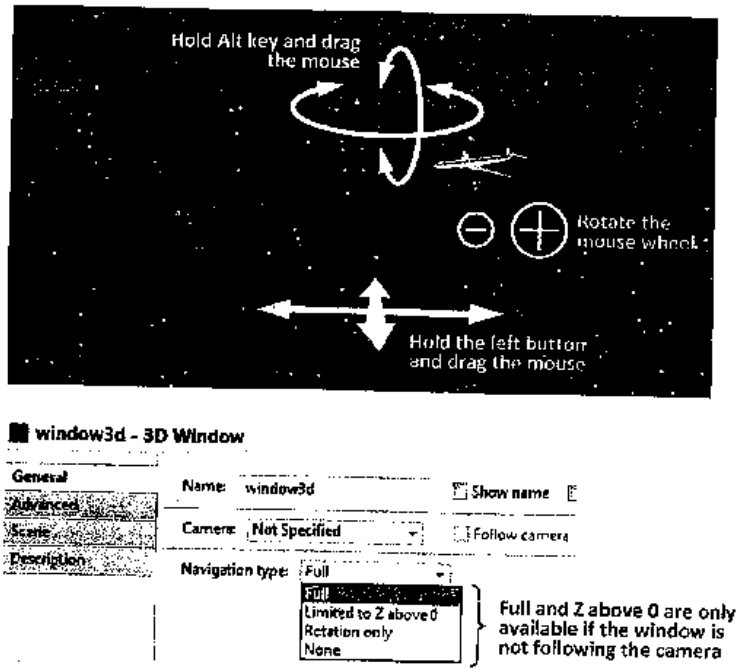


Figure 14.17 Navigation in the 3D scene

Keep in mind that, for performance optimization purposes, very distant objects are not displayed by the 3D rendering engine.

Sometimes you may wish to limit navigation. For example, you may not want the user to zoom in and out or to look below the ground level. You can limit navigation by choosing the appropriate **Navigation type** in the **General** page of the 3D window properties.

Multiple 3D views

While you have a single 3D scene per active object, you can have *multiple views of the scene* in multiple 3D windows.

▶ **Add a second 3D window to the simple model we created earlier**

1. Open the model from Example 14.1: "A very simple model with 3D animation".
2. Drag the second 3D window from the **3D** palette and drop it on the right of the first one.

3. Run the model and try to navigate in the second window so you see the people entering the shop through the shop door (see Figure 14.18).

Figure 14.18 Multiple views of the 3D scene

If the windows are not associated with cameras, in the beginning they all will display the scene from the same viewpoint. The viewpoint can then be changed individually for each window by using the mouse. To get different default viewpoints, you should link the widows with cameras. This is explained in the next section.

14.6. Cameras

Camera in the AnyLogic 3D toolset is a viewpoint that can be associated, both statically and dynamically, with a 3D window. There may be an arbitrary number of cameras in different active objects in the model. In particular, you can place a camera on a moving object and see things from its perspective. There may be several points of particular interest in your system such as known bottlenecks or sensitive equipment. You can assign a camera to each of them so as to quickly switch between these sensitive areas to assess the impacts of your simulation under different parameters or at different times.

Example 14.4: A very simple model with multiple 3D windows and cameras

We will use the first model we created in this chapter, and add the second 3D window and two cameras. The first one will be used as a default view for the first window, and the second one as the fixed unchangeable view for the second window.

The easiest way to tune the camera position is run the model, navigate a 3D window to the viewpoint you need, copy the viewpoint settings, and paste them into the camera properties in the editor.

▶ **Follow these steps:**

1. Open the model from Example 14.1: "A very simple model with 3D animation"
2. Reduce the width of the 3D window to leave space for another one.
3. Drag the second **3D window** from the **3D** palette and drop it on the right side of the first window.
4. Drag a **Camera** object from the **3D** palette and drop anywhere.
5. Run the model.
6. Adjust the view in the first 3D window so you can see the full scene at a nice angle.
7. Right-click the 3D window and choose **Copy camera location** from the context menu (see Figure 14.19).
8. Return to the model editor and select the camera. Press the **Paste** button in the camera properties (see Figure 14.19). The camera will jump to a new location, and its rotation angles will also change.

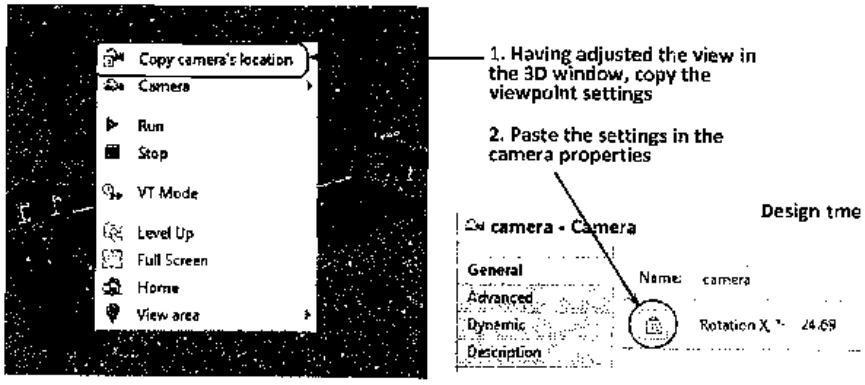


Figure 14.19 Tuning the camera position

9. Select the first 3D window. In its properties, set the camera to the **camera** you have created. Do not select the **Follow camera** checkbox. Set **Navigation type** to **Rotation only**.
10. Run the model. The window now shows the scene from a different viewpoint. You are able to adjust it slightly, but you cannot zoom in or out.
11. Add another camera and use the same procedure (steps 5-9) to set the view in the second 3D window to the view from the shop door, focusing on the people entering the shop.
12. In the properties of the second 3D window, select **Follow camera** and set **Navigation type** to **None**.
13. Run the model. Test navigation in both windows.

Example 14.5: Camera on a moving object

There are two ways you can move a camera. First, you can define dynamically changing coordinates and rotation in the camera **Dynamic** property page. Secondly, you can put the camera on a moving object. In this example, we will have an airplane with a camera that will fly over the US territory. We will use the function **setCamera()** of the 3D window to switch between the static and moving cameras.

▶ **First, create a 2D model:**

1. Create a new model.
2. Create a new active object class, **Airplane**. Select the **Agent** option in its properties.
3. Drag the **Airliner** object from the **3D objects** palette and place its center at (0,0). Rotate the airliner object by 180 degrees so that its nose looks to the right, and set its scale to 50%.
4. Open the editor of **Main**. Drag the **Airplane** object from the **Projects** tree to the canvas. Now **Main** has an embedded object **airplane**. Its embedded animation is located at (0,0). Move the airplane animation to (200,50).
5. Open the **Pictures** palette and drag the **USA Map** object to **Main**. Move the map so that its top left corner is located approximately at (200,50). Send the map to the back so that it gets below the airplane animation, as shown in Figure 14.20.
6. Open the editor of the **Airplane** class again and go to the **Agent** page of its properties. Write the following code in the **On arrival** section:
   moveTo(uniform(450), uniform(300));
   This code sends the airplane to a new random location within the US map once it has arrived at the previous one.
7. Below, in the same property page, set **Velocity** to **50**.
8. Copy the piece of code in **On arrival** and paste it in the **Startup code** field in the **General** property page. This way, we will cause the airplane to make the very first move.
9. Run the model and watch the airplane fly in 2D.

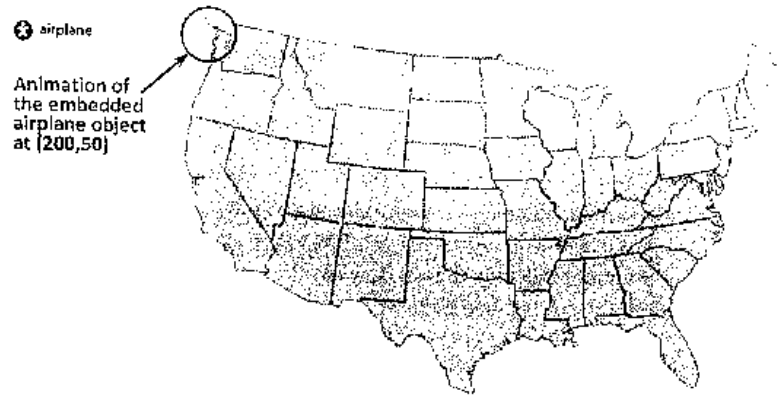


Figure 14.20 Map of the USA and the airplane object embedded in Main

▶ **Now, add the 3rd dimension to the model:**

10. Open the editor of **Main**, select the US map and select **Show in 3D scene** in its properties.
11. Select the animation of the embedded airplane, and set its **Z** to **30** on the **Advanced** property page. The airplane will fly at this altitude.
12. Drag the **3D window** from the **3D** palette and drop it below the map.
13. Run the model and see the airplane flying in 3D.

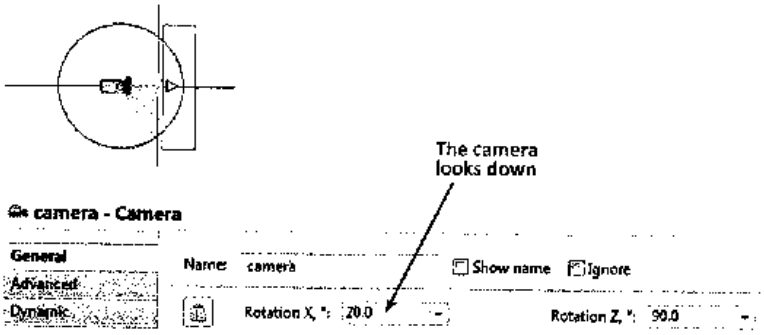


Figure 14.21 Camera on the airplane

▶ **Put a camera on the airplane:**

14. Open the editor of **Airplane** and drag the **Camera** object from the **3D** palette. Place the center of the camera slightly behind the tail of the aircraft and rotate it so that it looks toward the nose (see Figure 14.21).

15. On the **General** page of the camera properties, set **Rotation X** to 20 degrees. On the **Advanced** page, set **Z** to 20. This raises the camera above the airplane and bends it a bit down so that we can see the airplane itself.
16. Open the editor of **Main** and add a **Button** from the **Controls** palette. Label the button "Airplane camera". Type the following code in the button **Action** field:
    ```
    window3d.setCamera( airplane.camera, true );
    ```
    This code switches the 3D window to the airplane camera. The boolean argument **true** tells the window to follow the camera.
17. Run the model, watch the 3D animation, and then press the button. The 3D window now shows the view from the (virtual) point behind and above the moving airplane (see Figure 14.22).

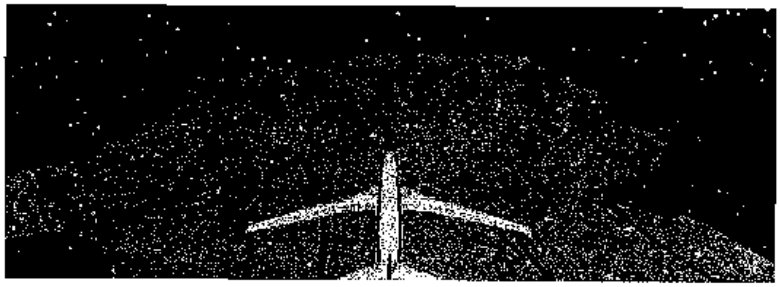


Figure 14.22 View from the airplane camera

We will now add a static camera and provide the ability to switch between the airplane camera and the static camera.

▶ **Add a static camera:**

18. Open the editor of **Main** and drag the **Camera** object from the **3D** palette in the bottom left corner of the map.
19. Add a second **Button** from the **Controls** palette and label it "Static camera". Type the following code in the button **Action** field:
    ```
    window3d.setCamera( camera, false, 500 );
    window3d.setNavigationMode( NAVIGATION_FULL );
    ```
20. Run the model and switch between the static and moving cameras.

The first line of code in the button action switches the 3D window to the static camera. The middle argument **false** tells the window not to follow the camera, and the last optional argument defines a smooth 500 millisecond transition between the views (you may add the same smooth transition to the code of the first button). The second line of code restores the ability to fully navigate within the 3D space.

14.7. Lights

To render the 3D scene, the 3D engine must know the sources and types of *light*. By default, the AnyLogic 3D engine assumes the presence of ambient light – the light that is everywhere (has no particular source), has no color, no direction and does not fade over distance. You can define your own lights in the 3D scene that will replace the default light. The lights can be of different types (such as point, spot, directional) and can have customized color and attenuation. They can also be static as well as moving.

In total, there can be up to 6 lights per 3D scene, including the lights that come with the embedded objects.

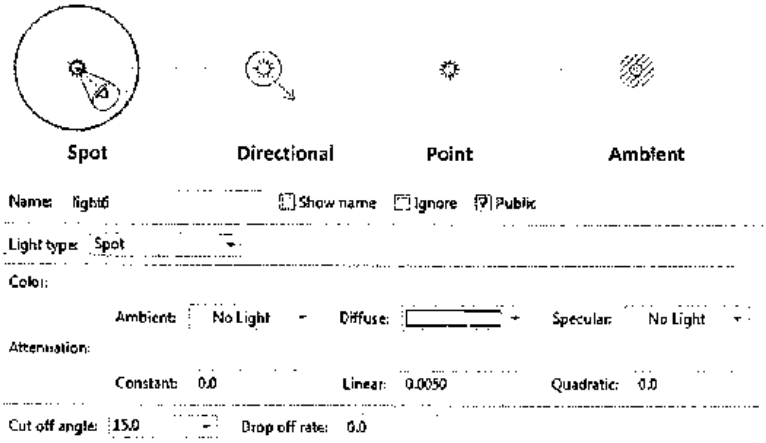


Figure 14.23 Light types and properties of a spot light

The **Light** object is located in the **3D** palette. In the properties, you can choose the type of light. The available types are:

- *Ambient* – ubiquitous light that has no particular source and no direction, like the light on a cloudy day. The default light that is assumed in the scene without any light objects is ambient gray.
- *Directional* – light from a source that is so far away that the rays are parallel, like sunlight.
- *Point* – light from a point source that shines evenly in all directions.
- *Spot* – light from a point source that shines only within a cone of a given angle.

A light of any type has three color components:

- *Ambient* – the color of the ubiquitous component of the light, if any.
- *Diffuse* – the color of the direct light that falls on the object surfaces.
- *Specular* – the color of the light reflected from the object surfaces.

All light color components interact with the object colors. If a light component is white or gray (evenly includes the whole spectrum), the objects will show in their natural colors. If, for example, the light color is pure red and the object color is pure blue, the object will not be visible. The purple object in the red light will show dark red, because only the red component of its color will interact with the light. Figure 14.24 shows six boxes in the directional light with different color components.

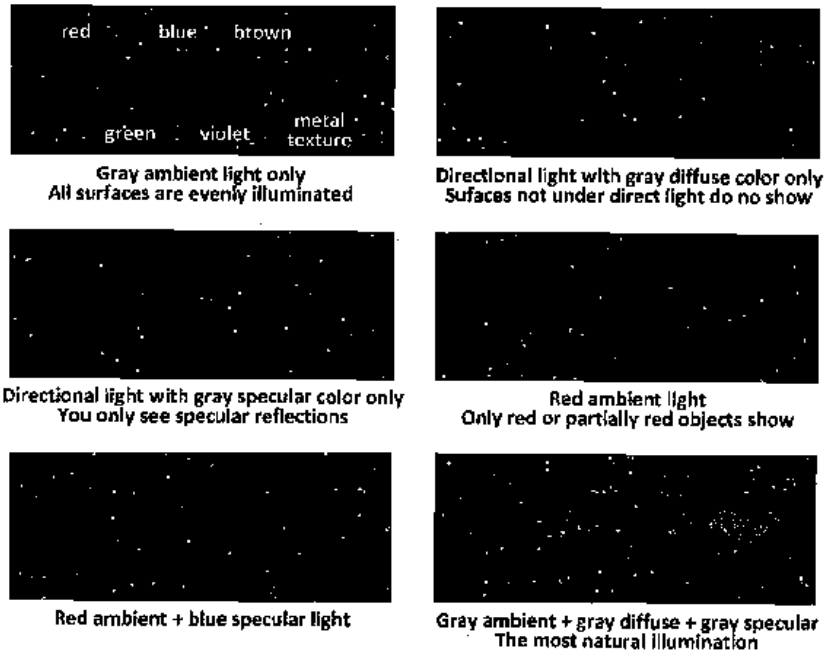

red
blue
brown
green
violet
metal texture
Gray ambient light only
All surfaces are evenly illuminated
Directional light with gray diffuse color only
Sufaces not under direct light do no show
Directional light with gray specular color only
You only see specular reflections
Red ambient light
Only red or partially red objects show
Red ambient + blue specular light
Gray ambient + gray diffuse + gray specular
The most natural illumination

Figure 14.24 Six boxes in the light with different color components

Point and spot lights have the **Attenuation** property. *Attenuation* is the effect of light diminishing over distance. Constant attenuation means the light intensity is totally unaffected by distance. The intensity of the light with linear attenuation is inversely proportional to the distance from the light source, and quadratic attenuation means a very sharp decline of light.

Example 14.6: Examples of Lights in 3D Scene

We will create a simple 3D scene with a road and a building with a gate. We will illuminate the road with streetlights and put an indoor spotlight inside the building. We will also create a car with headlights on it, and let it drive along the road and into the building.

▶ **Create a static 3D scene**

1. Create a new model.
2. Use the **3D** palette to create a scene, as shown in Figure 14.25. Add some men in black suits (office workers from the **3D Objects** palette) to the corner of the building.
3. Add a 3D window, set its **Background color** to **black**, and run the model. Remember how the scene looks like in the default light.

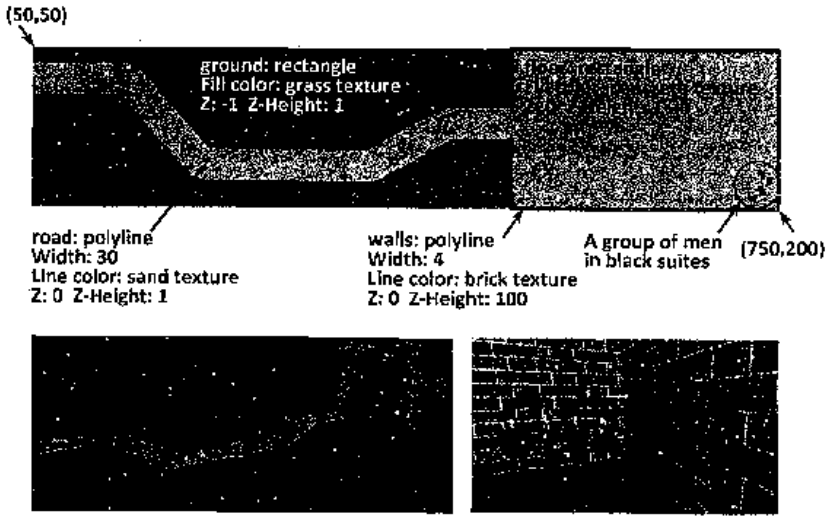


Figure 14.25 The 3D scene and its 3D view in the default light

▶ **Add static lights**

4. Add a **Light** object from the **3D** palette and place it anywhere. Choose the **Ambient** light type, set the **Ambient** color component to **darkGray**, and reset two other components.
5. Run the model again. The scene is now much darker, as we have replaced the default light with very dark ambient light.
6. Add two more lights of the **Point** type and place them nearby the road (see Figure 14.26). Set their **Diffuse** color component to **darkGoldenRod**, and reset two other components. Set the **Quadratic attenuation** to **0.0002** and the

Constant and **Linear** attenuation to **0**. Set the lights' **Z** to **50** on the **Advanced** property page.

7. Run the model and view the scene. You can see the road lit by the two streetlights. Quadratic attenuation creates the effect of light locality.
8. Add another light, this time of the **Spot** type, and place it in the bottom left corner of the building. Leave only the **Diffuse** color component set to **slateGray** and set the **Cut off angle** to 45 (this makes the light cone wider). In the **Advanced** property page set its **Z** to **50**, and **Angle X** to **55** (this rotates the light code downward).
9. Run the model and see the cone of the spotlight inside the building.

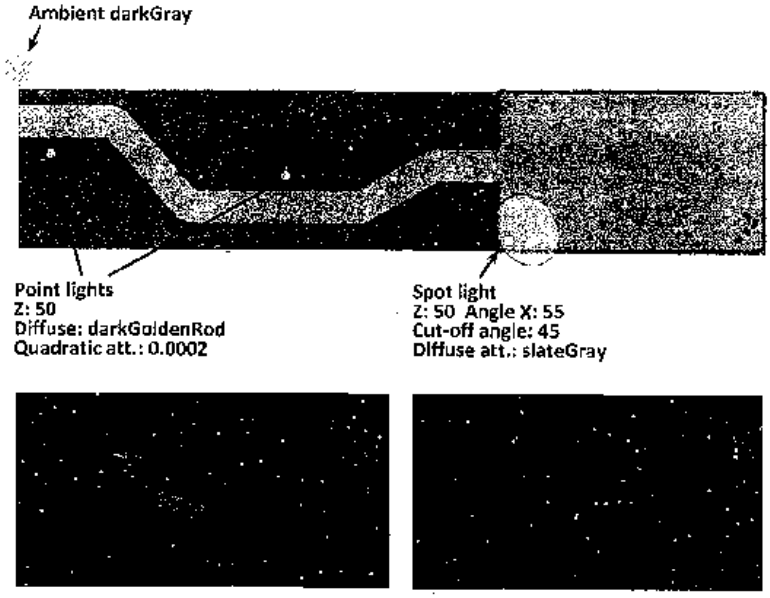


Figure 14.26 The 3D scene with the custom lighting

▶ **Add a car driving with headlights**

10. Add a new active object class to the model and call it **Car**. Make the **Car** an agent by selecting the corresponding checkbox.
11. Drag the **Car** object from the **3D objects** palette and drop it near the coordinate origin.
12. Add a light of the **Spot** type, place it at the car windshield, and rotate it so that it looks forward (see Figure 14.27). Set the **Diffuse** color to **lightSteelBlue**. On the **Advanced** properties page, set **Z** to **10** and **Angle X** to **5**.
13. Open the editor of **Main** and drag the **Car** object from the **Projects** tree. This creates an embedded object **car**, and its presentation appears on **Main**.

14. In the **Startup code** of Main, write:
    car.moveTo(670, 165, road);
    here, **road** is the name of the road polyline.
15. Run the model.

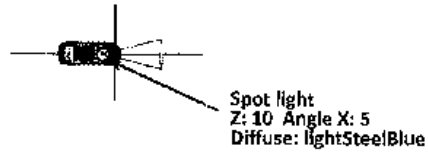


Figure 14.27 Car agent with a headlight driving in a 3D scene

Chapter 15. Randomness in AnyLogic models

The world we live in is non-deterministic. If today it took you 35 minutes to get to work, tomorrow it may be an hour, so when you set off in the morning you never know exactly how long it will take. You may know that on Friday evening on average 30 people come to your restaurant for dinner, but the time the first customer comes in tells you nothing about when you should expect the next one. John Smith who works for you as a salesman may have excellent skills, but when he is dealing with a particular prospect you never know for sure if he will be able to close the deal. When your company starts a new R&D project, you always hope it will bring you revenue in the end, but you always know it can fail. If you contact a person with flu you can get infected or you can successfully resist it. You are alive today, but nobody knows if you will be alive tomorrow.

Uncertainty is an essential part of reality and a simulation model has to address it to reflect this reality correctly. The only way of doing that is to *incorporate randomness into the model*, i.e. include the points that would give random results each time you pass them during the model execution.

This chapter describes possible ways to create sources of randomness in the model; namely the probability distributions, the random number generators and where and how you can use them in different kinds of models.

15.1. Probability distributions

Suppose you are modeling a business process in a bank, in particular the operation of opening a new bank account. You know from your observations that it takes a minimum of 10 minutes, most likely 20 minutes, and a maximum 40 minutes, but you did not bother to make serious measurements. How do you model the delay time associated with this operation? AnyLogic offers you a set of *probability distribution functions* that will return random values distributed according to various laws each time you call them. For the purposes of this example we can use the function **triangular(min, mode, max)** with these parameters:

```
triangular( 10, 20, 40 )
```

If you call that function several times you will get a sequence of results like these:

Result	11.555	18.592	30.945	24.867	21.346	31.423	22.741	28.350

and so on. As you can see, the results appear to be random and more or less consistent with your observations. To explore the function **triangular()** more

thoroughly you can call it very many times and build a histogram of the results distribution. The histogram will look like the one in Figure 15.1.

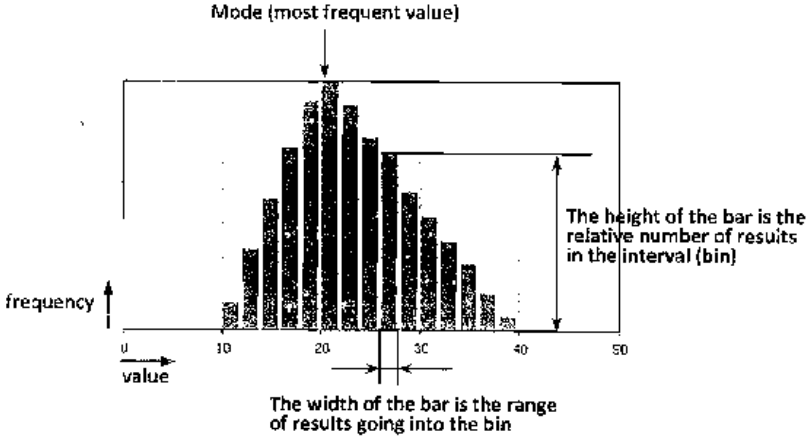


Figure 15.1 The distribution of 10,000 results returned by the function triangular(10, 20, 40)

The shape of the distribution (also called the *probability density function, PDF*) is a triangle with a minimum at 10, a maximum at 40 and a peak at 20 – the most frequent value also known as the *mode*. Indeed, the function **triangular(min, mode, max)** draws its results from the Triangular probability distribution with a given minimum, maximum, and most frequent values. We recommend using the triangular distribution function if you have limited sample data as in our bank example.

Depending on the type of your model you put a call to a probability distribution function, for example, into the **Delay time** parameter of a **Delay** or **Service** object, or into the **Timeout** field of a transition exiting the corresponding state, see Figure 15.2. Each entity passing the object (or each agent coming to the state) will get a new sample of the distribution.

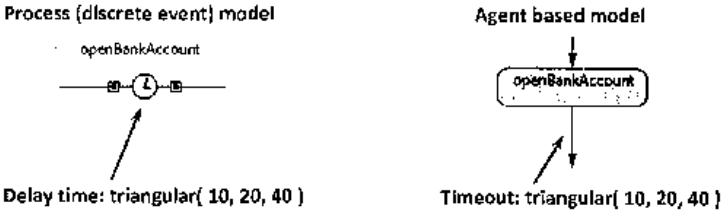
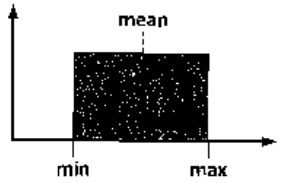
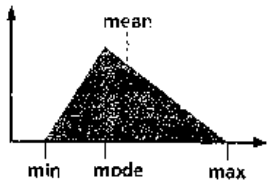
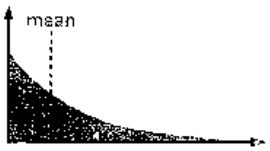
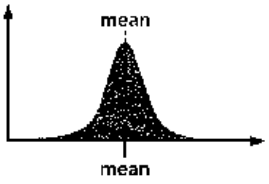
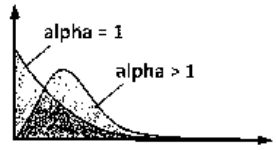
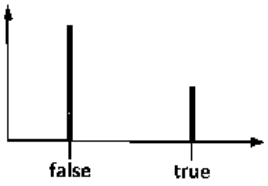


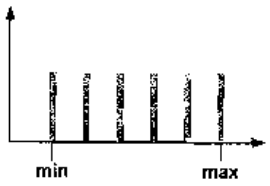
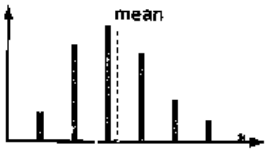
Figure 15.2 Using a probability distribution to model the duration of opening a bank account

Probability distribution functions

All randomness in AnyLogic models is based on calls to probability distribution functions. In total, AnyLogic supports about 25 distributions and offers over 50 corresponding functions. In the Table below we describe the most frequently used distributions. The complete information can be found in *AnyLogic Classes and Functions, API Reference* (The AnyLogic Company, 2013).

Distribution name, PDF form, and AnyLogic functions*,**	Primary use
Uniform  **uniform()** **uniform_pos()** **uniform(max)** **uniform(min, max)**	You know the minimum and the maximum values but have zero knowledge about how the values are distributed in between (i.e. you do not know if there are any values more frequent than others and assume a constant likelihood of a value being in any place between min and max). Used, for example, to generate coordinates of objects that are evenly spread over a rectangular area.
Triangular  **triangular(min, max)** **triangular(min, max, mode)** **triangular(min, max, left, mode, right)**	You know the minimum, the maximum, and have a guess about the most likely (modal) value. Used, for example, for service times, travel times, or, in general, for the duration of operations in conditions of limited sample data (too few samples to build a meaningful distribution shape).

Exponential  exponential() exponential(lambda) exponential(lambda, min) exponential(min, max, shift, stretch)	Describes the times between events in a Poisson process, i.e. when events occur independently at a constant average rate. Used as the inter-arrival time for input streams of customers, parts, calls, orders, transactions or failures in process models. In agent based models it is used as timeout for rate transitions (see Section 7.3) that model independent events in agents that are known to occur at a certain global average rate.
Normal  normal() normal(sigma) normal(sigma, mean) normal(min, max, shift, stretch)	Gives a good description of data that tend to cluster around the mean. For example, the height of an adult male person, the observation error in an experiment, etc. Note that the normal distribution is unbounded on both sides, so if you wish to impose limits (e.g. to avoid negative values) you have to either use its truncated form, or use other distributions such as Lognormal, Weibull, Gamma or Beta.
Gamma  gamma(alpha, beta) gamma(alpha, beta, min) gamma(min, max, alpha, shift, stretch)	A distribution bounded on the lower side. If **alpha** (the shape parameter) is 1 it reduces to the exponential distribution; for larger values of **alpha** it starts at 0, then has a peak and decreases monotonically. Used to model, for example, lifetimes, lead times or personal income data.
Random boolean  randomTrue(p) randomFalse(p)	Used to make a random decision between two alternatives with a given probability. For example, in process models used to divide the flow of entities into two, for example, economy and business class passengers or regular and urgent orders. In agent based models may be used in transition branches to model, for example, success or failure, and so on.

Discrete uniform  uniform_discr(max) uniform_discr(min, max)	Used to model a finite number of outcomes that are equally probable, or when you have no knowledge about which outcomes are more likely to occur. Example: a person (an agent) chooses a friend to communicate an idea. Note that both the minimum and maximum values are included in the set of possible results, so a call of uniform_discr(3, 7) may return 3, 4, 5, 6, or 7.
Poisson  poisson(lambda) poisson(min, max, mean, shift, stretch)	Discrete distribution describing the number of events occurring in a fixed period of time if these events occur independently and at a constant rate (lambda) Used to model, for example, the number of defects in a product or the number of calls in an hour, etc.

* There are many more probability distribution functions in AnyLogic. The full list with descriptions can be found in AnyLogic Help: *AnyLogic Classes and Functions* (The AnyLogic Company, 2013).

** Note that each distribution function has also a form name(..., Random r) which enables you to use a custom random number generator (see Section 15.3).

As you can see, in general there may be more than one function for a distribution: a short form that assumes default parameter values, and longer forms that allow you to tune the distribution for your particular problem. For example:

- normal() – the Normal distribution with mean at 0 and sigma (standard deviation) = 1
- normal(sigma) – mean at 0, custom standard deviation
- normal(sigma, mean) – both mean and standard deviation can be customized
- normal(min, max, shift, stretch) – same as above (shift is mean, stretch is sigma), but truncated to return values between min and max.

The latter form is provided for compatibility with Vensim™. *Truncation* of a distribution is performed in the following way: a draw from the original (not truncated) distribution is made, if the sample is outside the [min..max] interval, another try is made and so on until the sample is within the specified bounds.

Distribution fitting

If you are choosing the distribution in a state of limited information you can follow the advice in the previous section. However, if you have a data set (a set of observations) which well characterizes the random values in the system you are modeling, you can choose the right distribution by *fitting* it to the data set. *Distribution fitting* is the process of finding the analytical formula of a distribution that describes the random value as closely as possible to a given data set. There are various fitting heuristics and goodness-of-fit tests (e.g. the *Kolmogorov-Smirnov test*) ("Kolmogorov–Smirnov test", n.d.) and a number of software packages that would automatically perform distribution fitting and suggest one or several analytical distributions.

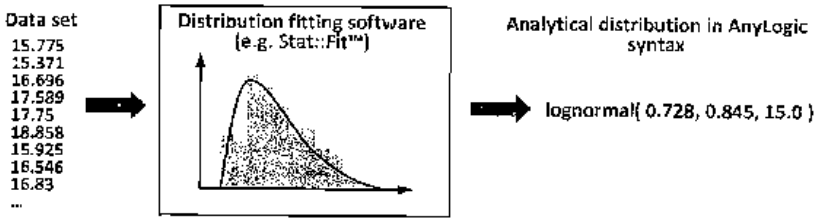


Figure 15.3 Input and output of distribution fitting software

Distribution fitting software would typically give you comprehensive statistics on your data set, display its histogram along with the PDF of the fitted distributions and rank the distributions according to several goodness-of-fit tests.

Should you be choosing distribution fitting software, it is recommended to use the one that directly supports the syntax of AnyLogic probability distribution functions, e.g. Stat::Fit (Geer Mountain Software Corporation, 2002). The output of such software can be directly copied into an AnyLogic model.

Custom (empirical) distributions

It may happen that no standard probability distribution can well fit the set of observations. In this case you can create a *custom* (also called *empirical*) distribution and use it in your model. As a first step you should build the PDF of your data set in the form of a histogram. This can be done using any statistical package, any distribution fitting software (which typically supports exporting in the form of an empirical distribution), or even an AnyLogic Histogram data object. Then you should construct an instance of the `CustomDistribution` class giving it the PDF as a parameter, for example in the form of a table function. After that you can draw random values from your distribution by using the function `get()`.

▶ **To create an empirical distribution from a data set:**

1. Process your data set and obtain its histogram in the following form:

Interval bounds	Frequencies (number of samples in the interval)
Start of the first interval	Frequency of the first interval
Start of the second interval	Frequency of the second interval
...	...
Start of the last interval	Frequency of the last interval
End of the last interval	0

2. Open the **General** palette and drag the **Table function** object to the canvas. Set the name of the table function to **customPDF**.
3. Copy the histogram data in the text form to the clipboard.
4. In the **General** property page of the **customPDF** table function press the **Paste from Clipboard** button located between the table and the graph.
5. Check the graph shape. Leave the **Interpolation type** of the table function set to **Linear** (you can also use **Step** interpolation if you wish).
6. Drag the **Variable** object from the **General** palette to the canvas. Set its name to **customDistribution**.
7. In the **General** page of the **customDistribution** properties set:
   Type: Other: CustomDistribution
   Initial value: new CustomDistribution(customPDF, getDefaultRandomGenerator())
8. The custom distribution is ready to use. To draw a value call:
   customDistribution.get()

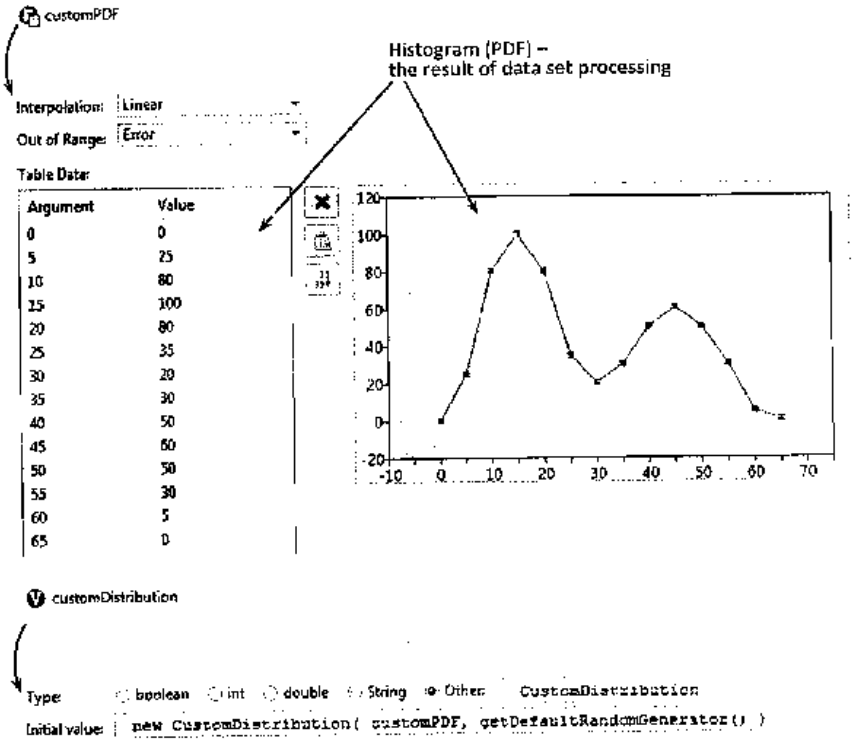


Figure 15.4 Two objects used to construct a custom (empirical) distribution

A random number generator (see Section 15.3) should be provided either in the constructor of the **CustomDistribution** (as in the example above), or as a parameter of the **get()** method. The custom distribution constructed from the table in Figure 15.4 will have a PDF as in Figure 15.5 below.

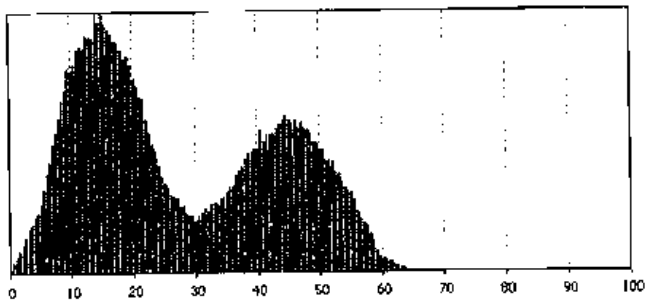


Figure 15.5 Histogram of 200,000 samples generated by the custom distribution

15.2. Sources of randomness in the model

There are stochastic and deterministic models. A *Deterministic model* has no internal randomness and, being run with the same set of input parameters, always drives the system through the same trajectory (the sequence of state changes) and gives the same output results. A *Stochastic model* has internal sources of randomness, and each run (even with the same parameters) may give a different trajectory and output.

In general there are more stochastic models than deterministic, especially among process (discrete event) models and agent based models. System dynamics models are mostly deterministic. This is explained by abstraction level: process and agent based models typically deal with individual objects, whose behavior has variations, while system dynamics deals with aggregates of large numbers of objects where individual variations are replaced (consumed) by averaging.

For a stochastic model you may need to perform multiple runs to get a meaningful picture of the system's behavior. The deterministic models are also often run multiple times with random variation of input parameters. A series of simulation runs with any kind of randomness (internal, at the level of input parameters, or both, see Figure 15.6) is called *Monte Carlo simulation*. The results of Monte Carlo simulation are statistically processed and typically have the form of probabilities, histograms, scatter plots or envelope graphs, etc.

Alternatively to performing multiple runs you can explore a stochastic model in a single run if each point of randomness is passed many times in a loop and the model enters a stochastically stable mode.

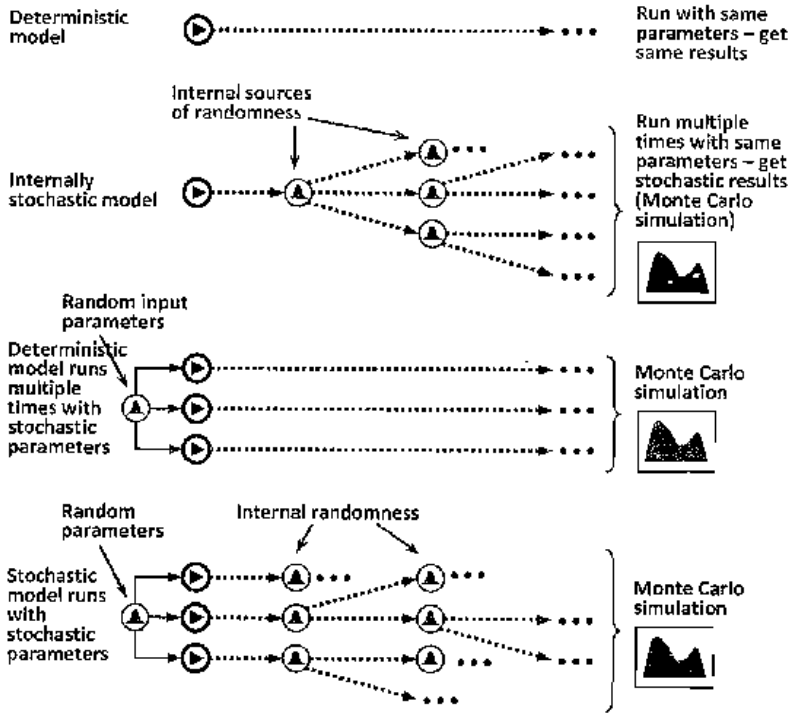

Deterministic model
Run with same parameters – get same results
Internal sources of randomness
Internally stochastic model
Run multiple times with same parameters – get stochastic results (Monte Carlo simulation)
Random input parameters
Deterministic model runs multiple times with stochastic parameters
Monte Carlo simulation
Random parameters
Internal randomness
Stochastic model runs with stochastic parameters
Monte Carlo simulation

Figure 15.6 Sources of randomness inside and outside the model. Monte Carlo simulation

In this section we focus on internal sources of randomness in different types of AnyLogic models.

Randomness in process models

Randomness is an essential part of virtually any process model, be it a model of a manufacturing site, call center, warehouse, hospital, airport or a bank. The durations of operations, the arrivals of clients, orders or patients, the human decisions and errors, the equipment failures, the delivery times, etc. all vary randomly from one instance to another. Therefore the process flowcharts typically contain plenty of calls to probability distribution functions.

By default, the fundamental process flowchart objects of the Enterprise Library have built-in randomness. These are (see Figure 15.7):

- Source: generates new entities with exponentially distributed inter-arrival time.

- **Delay** (and **Service** object based on it): has triangularly distributed delay time.
- **SelectOutput** (and its 5-exit version **SelectOutput5**): routes entities to different outputs with equal probability.

This means that, unless you explicitly eliminate the randomness from these objects, any process model you build is stochastic.

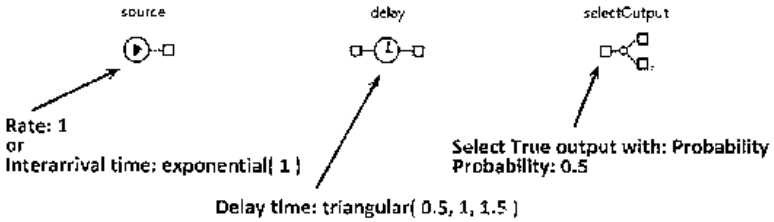


Figure 15.7 Randomness in the Enterprise Library objects

You may have noticed that if you run the simplest queuing model built of four objects **Source**-**Queue**-**Delay**-**Sink** with default parameters for a longer period of time, **Queue** overflow will occur. This is caused by the fact that mean values of entity inter-arrival time and delay time both equal 1; see Figure 15.7. In these conditions the length of the queue has no limited mean.

Other typical sources of randomness in process models are randomly distributed properties of entities and resource units. For example, in the model of a call center an entity class **PhoneCall** may have a field **complexity** with a randomly assigned value and the call center operators (resource units of class **Operator**) may have different random skills in different groups; see Figure 15.8. Then the time it takes an operator to answer the call and the algorithm of call redirection may depend on the complexity and skill values.

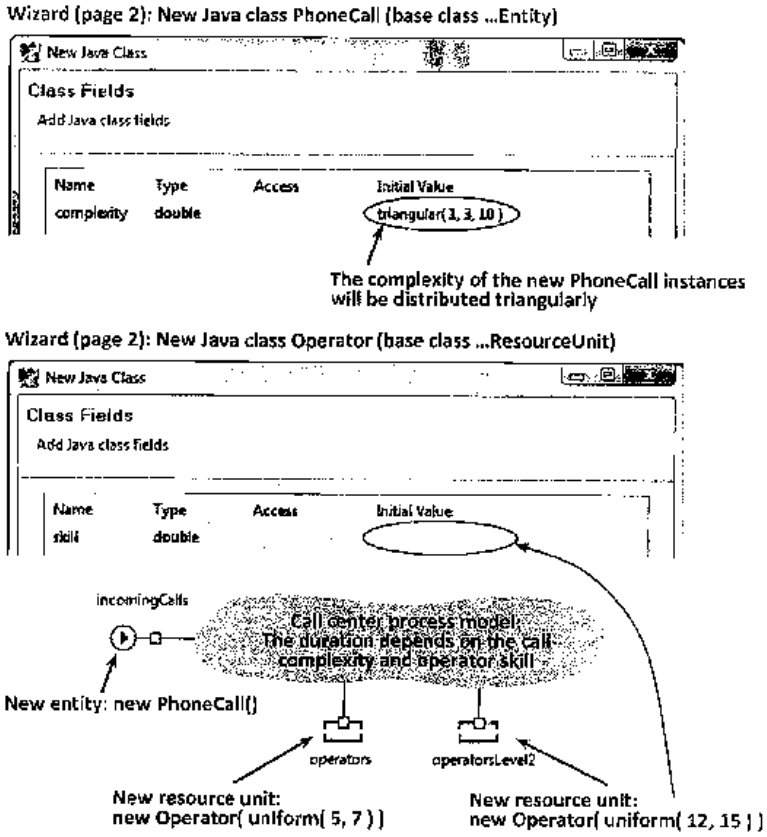


Figure 15.8 Random properties of entities and resource units in a call center model

Randomness in agent based models

As well as process models, most agent based models are stochastic too. In particular, randomness may be present in:

- The initial locations of the agents (if space is used)
- The network of agent connections
- The properties of agents
- The agent behavior; in particular in agent communication

A random layout is frequently used to evenly distribute the agents over a rectangular area; this is done using uniform distribution. **Random, Small world** and **Scale free** networks are constructed using link probabilities. Probability distributions are often used to set up the randomly distributed parameters over a population of

agents, like the Income parameter in Figure 15.9. While communicating with other agents a random friend or any random agent may be chosen. The transitions in the agents' statecharts may fire at random times and have nondeterministic results.

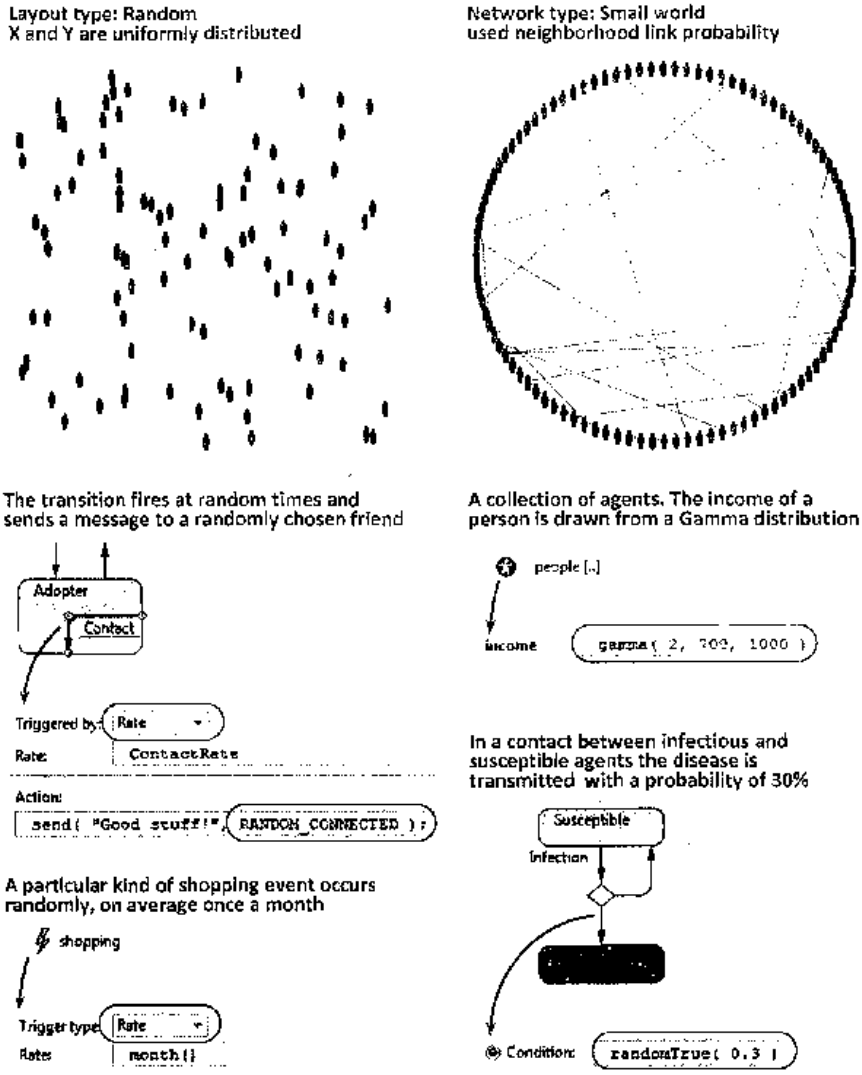


Figure 15.9 Randomness in agent based models

Example 15.1: Agents randomly distributed within a freeform area

Suppose in an agent based model the agents need to be randomly distributed in an area bounded by a closed curve or a polyline (which may mean a city limit, for example). As the standard layouts only distribute agents over rectangular areas or rings only, you will need to create your own layout.

▶ **Follow these steps:**

1. Press the **New** button on the toolbar. In the **New model** wizard enter the model name and on the next page choose the **Use template to create model** option and the **Agent Based** model template. Press **Next**.
2. On the next page of the wizard leave the default settings and press **Finish**. A new agent based model is created and the editor of its **Main** object opens.
3. Select the **environment** object and open the **Advanced** page of its properties. Set the **Layout type** to **User defined**.
4. Open the **Presentation** palette, double-click the **Curve** object and draw a curve like the one in Figure 15.10 by clicking at node points. Use double-click to finish drawing. Make sure the curve is fully on the right and fully below the presentation of the agent and that its width is less than 550, and its height is less than 400.
5. On the **General** page of the curve properties check the checkbox **Closed curve**.
6. Click the empty space of the editor to display the properties of the **Main** object.
7. On the **General** page write the following code in the **Startup code** field:

```
for( Person p : people ) { //for each agent
  double x, y;
  do {
    x = uniform( 550 );
    y = uniform( 400 );
  } while( ! curve.contains( x+200, y+50 ) ); // (200,50) is the origin of agents coords
  p.setXY( x, y ); //set the coordinates of the agent
}
```

8. Run the model. All agents should be randomly distributed across the area bounded by the curve.

In the startup code we are setting the initial location for each agent. We first generate a pair of random numbers `x` and `y` so that `x` is within 0..550 and `y` is within 0..400. Then we test if the point (x,y) is contained in the curve. (As the agent coordinate origin is where the agent presentation is located (the wizard places it at 200,50) and the `contains()` method accepts the global coordinates, we need to shift them by adding 200 and 50 correspondingly.) If the random point is not contained inside the curve, another point is generated. Sooner or later the point will be within

the curve bounds. Once this happens, we set up the agent coordinates and proceed to the next agent. The performance of this algorithm (i.e. the percent of successful tries) depends on the ratio of the area inside the curve to the enclosing rectangular area where we generate points. The same method would also work for shapes other than a curve.

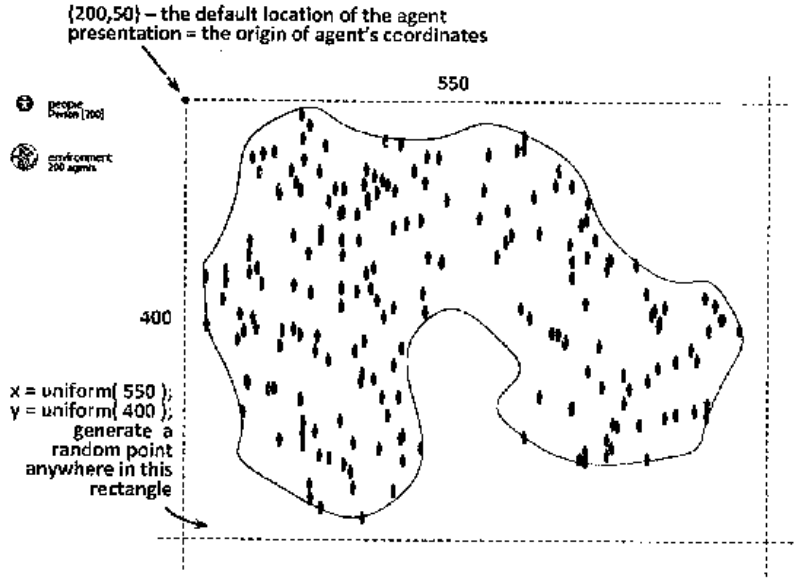


Figure 15.10 Agents randomly distributed within an area bounded by a curve

Example 15.2: Agents randomly distributed over a finite set of locations

Another similar task is to randomly distribute agents over a finite set of locations; for example seats in a movie theater or in an airplane. In this example we will use a discrete uniform probability distribution to generate a location. To define the set of locations we will use the points of a polyline.

▶ **Follow these steps:**

1. Press the **New** button on the toolbar. In the **New model** wizard enter the model name and on the next page choose **Use template to create model** option and the **Agent Based** model template. Press **Next.**
2. On the next page of the wizard leave the **Initial number of agents** set to **25** and press **Finish.** A new agent based model is created and the editor of its **Main** object opens.
3. Select the **environment** object and open the **Advanced** page of its properties. Set the **Layout type** to **User defined.**

4. Open the **Presentation** palette, double-click the **Polyline** object and draw a curve like the one in Figure 15.10 by clicking at node points. Create at least 30 points. Use double-click to finish drawing.
5. Click the empty space of the editor to display the properties of the **Main** object.
6. On the **General** page write the following code in the **Startup code** field:

```
int n = polyline.getNPoints(); //total number of locations = number of points
boolean[] occupied = new boolean[n]; //remember which are occupied
int freeplaces = n; //number of free locations
for( Person p : people ) { //for each agent
  int freeindex = uniform_discr( freeplaces-1 ); //random free location
  int ptindex = 0; //we will look for the index of the polyline point, start at 0
  while( true ) {
    if( ptindex >= n )
      error( "All points are occupied. Cannot find a place for the agent" );
    if( !occupied[ptindex] ) //if location is free
      if( freeindex == 0 ) { //if this is the index we are looking for
        //place the agent there
        double x = polyline.getX() + polyline.getPointDx( ptindex ) - 200;
        double y = polyline.getY() + polyline.getPointDy( ptindex ) - 50;
        p.setXY( x, y );
        occupied[ ptindex ] = true; //mark location as occupied
        freeplaces--; //decrement the number of free locations
        break; //exit the while loop
      } else {
        freeindex--; //not yet at the desired free location – decrement index
      }
    ptindex++; //will look at the next point of the polyline
  }
}
```

7. Run the model. All agents should be randomly distributed over the points of the polyline.

The algorithm above is fairly long and differs from the one used in the previous example to find a point within the curve bounds. We could use a similar method; namely we could be directly generating the random index of the location, then test if it is free, and, if not, make another draw from the discrete uniform distribution. However, for a large number of locations densely occupied by agents it would be very inefficient: suppose you have already placed 9,990 football fans into a stadium with 10,000 seats. To place the next person you would need to make hundreds of tries before the result of `uniform_discr( 9999 )` gives you one of the free seats left. In our algorithm we are directly choosing a random seat among the 10 free seats.

Randomness in system dynamics models

A system dynamics model built of standard elements, i.e. stocks, flows and feedback loops has no internal randomness (is deterministic) unless you explicitly insert it into the model. When doing that please follow the guidelines below:

In AnyLogic each new call to a probability distribution function generates a new value. Therefore you should not use those functions in the formulas of system dynamics models (see Chapter 5): the numeric solver evaluates the formulas *several times in one time step* and will be confused if it gets different results.

To model values randomly changing in time in system dynamics models you should create a variable or SD constant and assign a random value to it at each time step (or less frequently, depending on the model logic) using, for example, a cyclic event (see Section 8.2).

Example 15.3: Stock price fluctuations in a system dynamics model

Suppose in a system dynamics model you wish to have a variable for a stock price that randomly changes every day. A daily change is random and is not larger than 2 in either direction.

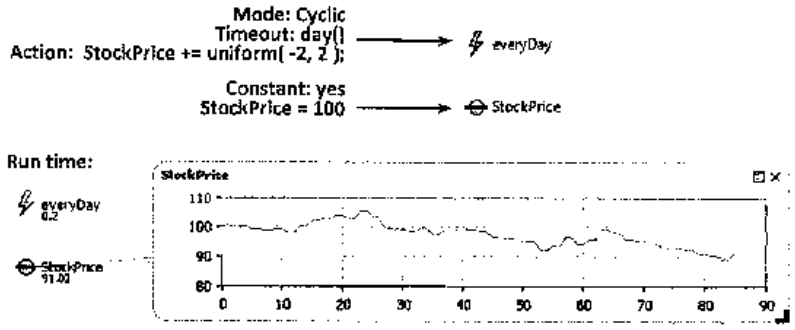


Figure 15.11 Random fluctuations of a stock price in a system dynamics model

▶ **Follow these steps:**

1. Open the editor of the active object. Open the **General** palette and drag the **Event** object to the canvas. Set the name of the event to **everyDay**.
2. In the **General** page of the event properties set **Mode** to **Cyclic** and set the **Recurrence time** to **day()**.
3. Open the **System Dynamics** palette and drag the **Flow Aux Variable** object to the canvas. Set the name of the variable to **StockPrice**.
4. In the **General** page of **StockPrice** check the **Constant** checkbox and set the value to **100**.

5. Open the **General** property page of the event **everyDay**. In the **Action** field write: `StockPrice += uniform( -2, 2 );`
6. Run the model. Click the **StockPrice** and watch the changes in the **Inspect** window.
7. You can now use the random variable **StockPrice** in any formula in the system dynamics model.

By marking the **StockPrice** as **Constant** we just tell AnyLogic that the value specified in its properties (in our case **100**) *should be treated as an initial value only and should not be treated as a formula and evaluated by the numeric solver.* Instead we are explicitly assigning a new value to the **StockPrice** every day by the event **everyDay**. (As an alternative to a constant system dynamics variable we could also use a variable.)

The uniform distribution was used as we do know the bounds of a daily change but do not know if any changes within those bounds are more likely than others.

Randomness in AnyLogic simulation engine

Besides the sources of randomness at the model level as discussed above, there is only one internal source in AnyLogic simulation engine, namely the ordering of simultaneous events. This topic is extensively addressed in the Chapter 8, "Discrete events and Event model object", here we would like to mention that:

- The engine uses the same default random number generator as the probability distribution functions do, and
- You can turn that randomness on and off:

▶ **To set the ordering mode for simultaneous events:**

1. Select the experiment and open the **Advanced** page of its properties.
2. Depending on what you want, check or uncheck the checkbox **Random selection mode for simultaneous events.**

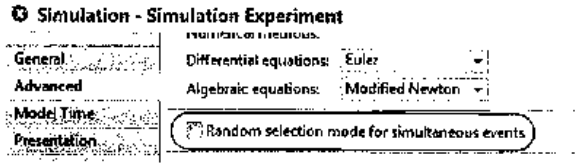


Figure 15.12 You can turn random ordering of simultaneous events on and off

15.3. Random number generators. Reproducible and unique experiments

The computer (or, at least, the processor executing a program) is a completely deterministic device: its next state is fully determined by the current state. So, when we talk about randomness in simulation models, did you ever wonder where that randomness comes from?

The truth is that, unless a software application accesses an external physical random number generator, there is no real randomness in it; however a computational random number generation may be used to create pseudo-randomness.

Random number generators

A *random number generator* (*RNG*) is a device that generates a sequence of numbers that lack any pattern, i.e. appear random ("Random number generator," n.d.). There are two types of RNGs: physical and computational. *Physical RNGs* have been known from ancient times: dice, coin flipping, shuffling of playing cards, roulette wheel and other techniques. The modern ones use atomic and subatomic physical phenomena, such as radioactive decay or thermal noise, or (like (Random .org, , 1998-2012)) atmospheric noise, or radio noise. Physical RNGs generate "true" random numbers, i.e. those that are completely unpredictable.

Computational RNGs are based on deterministic algorithms that produce long sequences of apparently random results, which are in fact completely determined by a shorter initial value, known as a *seed*, and are periodic. They are therefore called *pseudo-random number generators*. Pseudo-random number generators can be used instead of "true" ones in many applications; in particular, in the majority of simulation models. Moreover, their predictability feature is used to create reproducible stochastic experiments.

By default, all probability distribution functions in AnyLogic, the Enterprise Library objects, the random transitions and events, the random layouts and networks and the AnyLogic simulation engine itself – in other words, all randomness in AnyLogic, is based on the *default random number generator*. The default random number generator is an instance of the Java class **Random**, which is a *Linear Congruental Generator* (LCG). The LCG is one of the oldest and best known pseudo-random generators. It is very simple: the stream of random numbers is determined by the recurrence relation:

$$X_{n+1} = (aX_n + c) \bmod m$$

where a is a multiple, c is increment, and m is modulus. The period of LCG is at most m, and in the class **Random** $m = 2^{48}$. The initial value X_0 is a seed.

If for any reason you are not satisfied with the quality of **Random**, you can:

- Substitute your own RNG instead of the AnyLogic default RNG.
- Have multiple RNGs and explicitly specify which RNG should be used when calling a probability distribution function.

▶ **To substitute the default RNG with your own RNG:**

1. Prepare your custom RNG. It should be a subclass of the Java class **Random**, e.g. **MyRandom**.
2. Select the experiment and open the **General** page of its properties.
3. Select the radio button **Custom generator (subclass of Random)** and in the field on the right write the expression returning an instance of your RNG, for example:
   new MyRandom() or **new MyRandom(1234)**

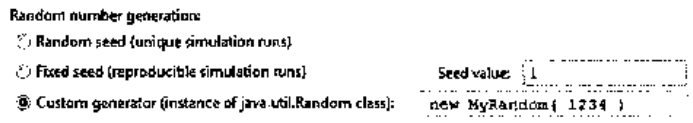

Random number generation:
Random seed (unique simulation runs)
Fixed seed (reproducible simulation runs)
Seed value: 1
Custom generator (instance of java.util.Random class):
new MyRandom(1234)

Figure 15.13 Setting a custom random number generator as default RNG

The initialization of the default RNG (provided by AnyLogic or by you) occurs during the initialization of the experiment and then before each simulation run.

In addition you can substitute the default RNG at any time by calling:

```
setDefaultRandomGenerator( Random r )
```

However you should be aware that before each simulation run the generator will be set up again according to the settings on the **General** page of the experiment properties.

▶ **To use a custom RNG in a particular call of a probability distribution function:**

1. Create and initialize an instance of your custom RNG. For example, it may be a variable **myRNG** of class **Random** or its subclass.
2. When calling a probability distribution function, provide **myRNG** as the last parameter, for example:
   uniform(myRNG) or **triangular(5, 10, 25, myRNG)**

If a probability distribution function has several forms with different parameters, some of them may not have a variant with a custom RNG, but the one with the most complete parameter set always has it.

The seed. Reproducible and unique experiments

Although pseudo-random number generators do not produce "truly random" streams of numbers, they have one feature which is very important in simulation modeling: having been initialized with a particular seed they generate exactly the same sequence of numbers each time. This enables you to create *reproducible experiments* with stochastic models, which would be impossible with a "true" RNG. Reproducibility, i.e. the ability to run the model along the same trajectory of state changes, is useful when you are debugging the model, or when you wish to demonstrate a particular scenario.

In AnyLogic you have two options for the standard RNG, see Figure 15.14: you can choose **Random seed** to run unique experiments, or **Fixed seed** to run reproducible experiments. The seed is set during the initialization of the experiment and then at the beginning of each simulation run (replication). If a custom RNG is provided, AnyLogic at those points just sets the default RNG to what is specified in the corresponding field.

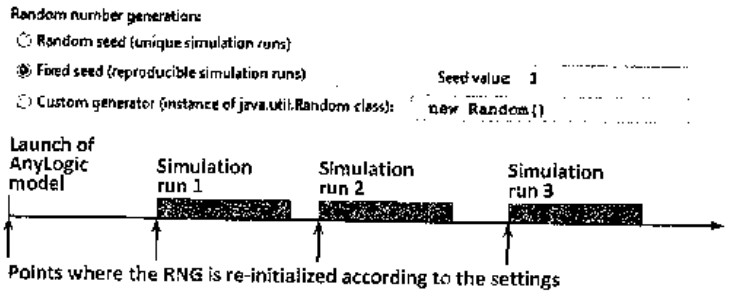

Random number generation:
Random seed (unique simulation runs)
Fixed seed (reproducible simulation runs)
Seed value: 1
Custom generator (instance of java.util.Random class):
new Random()
Launch of AnyLogic model
Simulation run 1
Simulation run 2
Simulation run 3
Points where the RNG is re-initialized according to the settings

Figure 15.14 AnyLogic RNG seed settings and the points when they are applied

A valid question is: where is the "random seed" taken from? When the random seed option is chosen, AnyLogic calls the default constructor of Java class Random(), which sets the seed of the random number generator to a value very likely to be distinct from any other invocation of this constructor. In the earlier versions of Java the computer system time was used as a seed in that case.

Example 15.4: Reproducible experiment with a stochastic process model

We will create the simplest queuing system: a source of entities, a queue, a delay with capacity 1, and a sink object. The model will have two sources of randomness: the arrival times of the entities and the delay time: under the default settings the inter-arrival times are distributed exponentially and the delay times triangularly. (See Section 15.2, subsection "Randomness in process models".) We will record the entity exit times and compare them for different runs.

▶ **Create and run the model with default seed settings (random seed)**

1. Press the **New** (model) button on the toolbar. In the **New model** wizard enter the model name and on the next page choose the **Use template to create model** option and the **Discrete Event** model template. Press **Next** and press **Finish** on the next page. A new process model is created and the editor of its **Main** object opens.
2. Select the **sink** object. In the **On enter** field of its properties write:
   traceln(time()); – each time the entity exits the model we will write the current time to the model log.
3. In the **Projects** tree select the **Simulation** experiment. On the **Model time** page of the experiment properties set:
   Stop: Stop at specified time
   Stop time: 20
4. Run the model up to completion. Look at the model log in the **Console** window of the AnyLogic model development environment: there should be about 20 records.
5. Press the **Stop** button on the model toolbar and run the model again. Another portion of records should appear in the log. The entity exit times are different from the ones in the first run.
6. Close the model window and run the model again. The output should again be different.

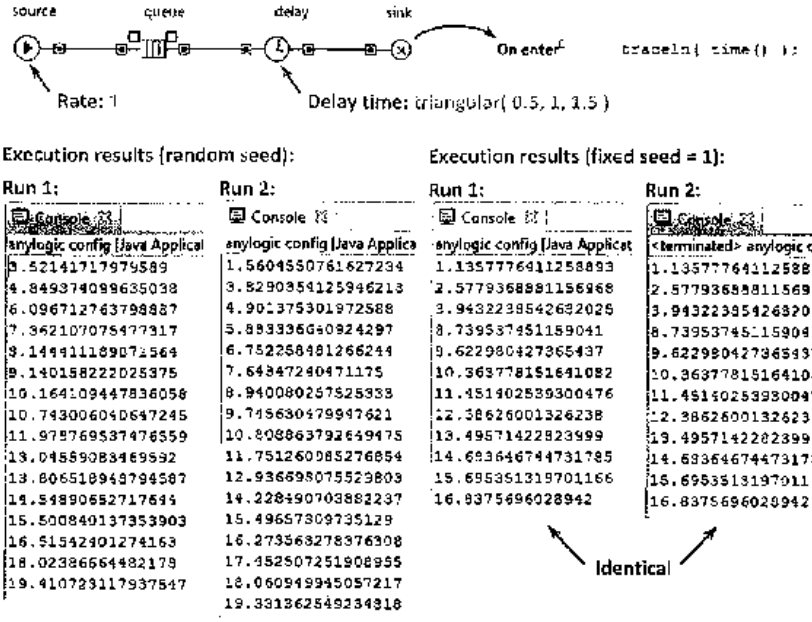


Figure 15.15 Unique and reproducible experiments with a stochastic process model

You see that under the default settings of AnyLogic RNG each run of the stochastic model is unique. Now let us change the RNG settings.

▶ **Change the RNG settings to run reproducible experiments**

7. In the Projects tree select the Simulation experiment. On the General page of its properties choose **Fixed seed (reproducible simulation runs)** and leave the default seed value.
8. Run the model several times and compare the outputs. They should be exactly the same.

The important thing is that the results will be the same every time and everywhere: you may export your model, send it to a client, or publish it as a Java applet on the web – and anybody who runs it will observe exactly the same behavior.

Chapter 16. Model time, date and calendar. Virtual and real time

Time is the central axis in the dynamic simulation models we are building. The models are full of various references to time: delays, arrival times, service times, rates, timeouts, schedules, dates, velocities, etc. This chapter explains what model time is and how the user can work with it.

16.1. The model time

Model time is the virtual (simulated) time maintained by the AnyLogic simulation engine. The model time has nothing to do with the real time or the computer clock (although you can run the model in a scale to real time), see Section 16.3).

In AnyLogic, the model time takes Java **double** type values (real numbers with double precision). The model clock is advanced in steps: while the engine is executing a discrete event model, the model time jumps from one event to another (see Section 8.1); if a continuous-time model is being executed, the time steps are typically smaller and have equal size.

As all events in AnyLogic are instantaneous and indivisible, the model time does not progress during event execution, no matter how long it takes to complete all the computations associated with the event.

The current model time can be obtained during execution of the model by calling:

- **double time()** – returns the current model time.

At the beginning of a simulation run, the time typically equals 0, although you can change it. You can set the *stop time*, the time at which you wish the simulation run to be terminated.

▶ **To set the start and stop times of a simulation run:**

1. Select the experiment and open its **Model time** property page.
2. Set the initial value of the model clock in the **Start time** field (by default, it is 0, and typically you do not need to change it).
3. If you wish the model to stop at a specific time, set **Stop** to **Stop at specified time** and enter the time in the **Stop time** field.
4. If you do not want the model to stop at a specific time, set **Stop** to **Never**.

Setting **Stop** to **Never** does not mean the simulation will run infinitely long. It can be terminated in a number of other ways: programmatically, using a stop condition, or when the engine detects there are no more dynamics left in the model.

If you are using calendar (see Section 16.2), you can set the stop date instead of the stop time.

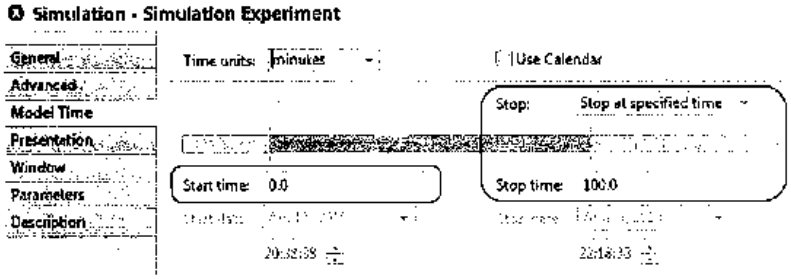


Figure 16.1 Setting the start and stop times

Time units

To establish the correspondence between the model time and real world time where the system being modeled lives, we need to define the *time units*. The type of time unit depends on the time scale of the activities you are modeling. For example, if you are modeling a call center where the call durations are measured in seconds or minutes, you may set the time units to seconds or minutes. If you are modeling a supply chain, where manufacturing and shipping times are measured in days, days would be the right choice. The expression **triangular(10, 12, 15)** used, e.g., in the **Delay** object, means a minimum of 10 days, a maximum of 15 days and most likely value of 12 days if day is the time unit.

▶ **To set the time units:**

1. Select the experiment and open its **Model time** property page.

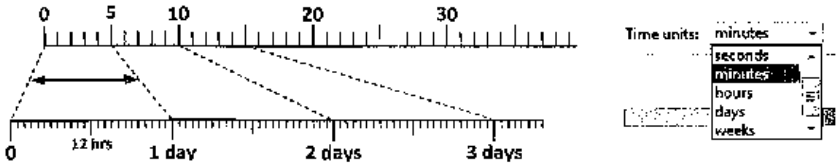


Figure 16.2 Setting the time units

The available time units are given in the Table below.

Time units	AnyLogic constant	Value
milliseconds	TIME_UNIT_MILLISECOND	1
seconds	TIME_UNIT_SECOND	1000
minutes	TIME_UNIT_MINUTE	60*1000
hours	TIME_UNIT_HOUR	60*60*1000
days	TIME_UNIT_DAY	24*60*60*1000
weeks	TIME_UNIT_WEEK	7*24*60*60*1000

AnyLogic offers the following API to work with time unit settings. (These are the methods of the engine; therefore, if they are called from an experiment or active object, they must be accompanied by the "**getEngine().**" prefix):

- **long getTimeUnit()** – returns the current time unit, namely the number of milliseconds in one time unit. If you are using one of the standard units, a constant from the Table above is returned.
- **setTimeUnit(long tu)** – sets the time unit to a given number of milliseconds (**tu**). If you are setting a standard time unit, you can use a constant from the Table.

You may have noticed that there are no time units such as months or years. This is because any time unit longer than a week is not a constant: a month may have 28, 29, 30 or 31 days, and a year can have 365 or 366 days. If you wish to have, say, months as the time unit in your model, then you have two options:

- Ignore the time unit settings and just think that 1 means one month. Then in the model, you should not use specific time functions such as **day()**, **week()**, etc. and you should not use the calendar.
- Set a custom time unit by calling
  getEngine().setTimeUnit(30 * TIME_UNIT_DAY);
  in the **Initial experiment setup** field on the experiment's **Advanced** property page. Then 1 in the model will mean 30 days, and you will be able to use the functions such as **week()** and **day()** and calendar, although the calendar months will obviously differ slightly from months based on time units and the longer the period, the bigger the error.

Developing models independent of time unit settings

Let's say the time unit in your model is hours. What if you need to schedule something to happen in 2 days? Or how would you define a duration of 5 minutes? Of course, you could write 48 and 5.0/60. But a much better solution is to use the

special functions that return the value of a given time interval with respect to the current time unit settings:

- **double millisecond()** – returns the value of a one-millisecond time interval.
- **double second()** – returns the value of a one-second time interval.
- **double minute()** – returns the value of a one-minute time interval.
- **double hour()** – returns the value of a one-hour time interval.
- **double day()** – returns the value of a one-day time interval.
- **double week()** – returns the value of a one-week time interval.

For example, if the time unit is hours, **minute()** will return 0.0166, and **week()** will return 168.0. Thus, instead of remembering what the current time unit is and writing 48 or 5./60, you can simply write **2*day()** and **5*minute()**. You can also combine different units in one expression: **3 * hour() + 20 * minute()**.

What is probably even more important about these functions is that the expressions using them are completely independent of the time unit settings: the expressions always evaluate to the correct time intervals. Therefore, we recommend always using multipliers such as **minute()**, **hour()**, **day()**, etc. in the numeric expressions that represent time intervals: this way, you can freely change the time units without changing the model.

16.2. Date and calendar

To use calendar in the model you need to tie the start point of the simulation to a particular date. This is also done on the experiment's **Model time** property page.

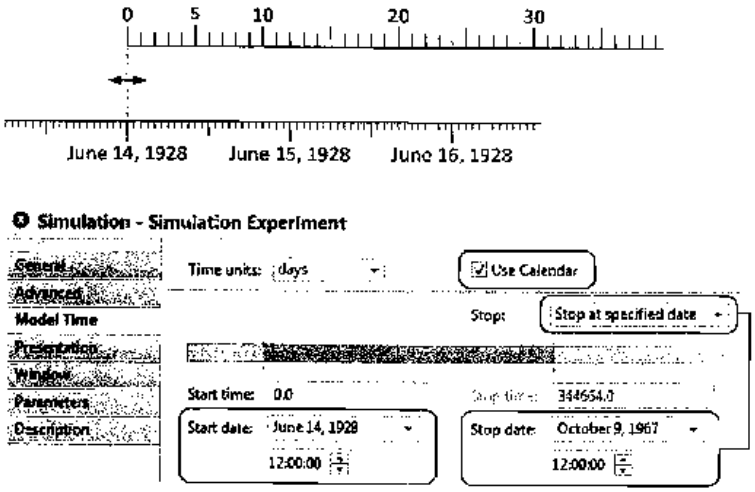


Figure 16.3 Setting the start and stop dates

▶ **To set the simulation start date:**

1. Select the experiment and open its **Model time** property page.
2. Check the checkbox **Use calendar**.
3. Use the **Start date** control to set the start date.

By default, the *start date* is set to the date when the model is created.

If **Stop** is set to **Stop at specified time**, the end date of the simulation will be:

Start date + Time unit * (Stop time – Start time)

Alternatively, you can set the stop date explicitly:

▶ **To set the simulation stop date:**

1. On the same property page, set **Stop** to **Stop at specified date**.
2. Use the **Stop date** control to set the stop date.

The stop time will automatically recalculate.

Most model elements accept time (**double** type values measured in model time units) as their parameters: event and transition timeouts, delay and interarrival times in the Enterprise Library objects, etc. Therefore, you need to be able to efficiently manipulate dates and convert dates into time values and vice versa. AnyLogic provides a rich API for that purpose.

Finding out the current date, day of week, hour of day, etc.

The date in AnyLogic is stored in the form of the Java class **Date**. **Date** is composed of the year, month, day of month, hour of the day, minute, second and millisecond. To find out the current date, you should call:

- **Date date()** – returns the current model date.

A number of functions return particular components of the current date (and all those functions also have the form with parameter **<function name>(Date date)**, in which case they return the component of a given, not current, date):

- **int getYear()** – returns the year of the current date.
- **int getMonth()** – returns the month of the current date: one of the constants **JANUARY, FEBRUARY, MARCH**, ...
- **int getDayOfMonth()** – returns the day of the month of the current date: 1, 2, ...
- **int getDayOfWeek()** – returns the day of the week of the current date: one of the constants **SUNDAY, MONDAY**, ...
- **int getHourOfDay()** – returns the hour of the day of the current date in 24-hour format: for 10:20 PM, will return 22.
- **int getHour()** – returns the hour of the day of the current date in 12-hour format: for 10:20 PM, will return 10.
- **int getAmPm()** – returns the constant **AM** if the current date is before noon, and **PM** otherwise.
- **int getMinute()** – returns the minute within the hour of the current date.
- **int getSecond()** – returns the second within the minute of the current date.
- **int getMillisecond()** – returns the millisecond within the second of the current date.

Consider a model of a processing center that operates from 9 AM to 6 PM on weekdays. The following function returns **true** if the center is currently open and **false** otherwise:

```
boolean isOpen() {
  int dayofweek = getDayOfWeek();
  if( dayofweek == SUNDAY || dayofweek == SATURDAY )
    return false;
  int hourofday = getHourOfDay(); //will be in 24-hour format
  return hourofday >= 9 && hourofday < 18;
}
```

Do not confuse the functions **hour()**, **minute()**, ... with the functions **getHour()**, **getMinute()**,... While **hour()** returns the duration of the hour in model time units, **getHour()** returns the current hour of the day.

Constructing dates. Converting the model date to the model time and vice versa

To create a **Date** object from its components (year, month, etc.), use the following function:

- **Date toDate(int year, int month, int day, int hourOfDay, int minute, int second)** – returns the date in the default time zone with the given field values.

To convert between a given model time and the model date, you can call:

- **Date timeToDate(double t)** – converts a given model time to the model date with respect to the start date, start time and model time unit settings; returns null if the time is infinity.
- **double dateToTime(Date d)** – converts a given model date to the model time with respect to the start date, start time and model time unit settings.

For example, to obtain the time interval (i.e., the number of model time units) between the two dates, you should write:

```
dateToTime( date1 ) – dateToTime( date0 )
```

Suppose you wish to find out the model time corresponding to 9 AM on the nearest Monday in the future. You can write the following function:

```
double timeOnNearestDayOfWeek( int dayofweek, int hour, int minute, int second ) {
  //obtain same time today
  double sametimetoday =
    dateToTime( toDate( getYear(), getMonth(), getDayOfMonth(), hour, minute, second ) );
  //obtain integer difference in days (may be negative or zero)
  int daydiff = dayofweek - getDayOfWeek();
  //correct difference if needed
  if( daydiff < 0 || daydiff == 0 && time() > sametimetoday )
    daydiff += 7;
  return sametimetoday + daydiff * day();
}
```

The function returns the model time corresponding to a given time of day on a given day of week nearest in the future; in particular, for Monday 9 AM, you should write:

```
timeOnNearestDayOfWeek( MONDAY, 9, 0, 0 )
```

Such functions may be used, e.g., in the timeout expressions of events and statechart transitions.

To find out how many days, months or years are contained between two given dates or model times, you can use the following two functions (one of the constants **YEAR**, **MONTH**, **WEEK**, **DAY**, **HOUR**, **MINUTE**, **SECOND**, **MILLISECOND** should be used as the **timeUnit**):

- **double differenceInDateUnits(int timeUnit, Date date1, Date date2)** – returns the number of given time units between the two dates.
- **double differenceInDateUnits(int timeUnit, double time1, double time2)** – returns the number of given time units between the two model times.

Specifying timeouts and delays in days, months, years

In simulation models, you sometimes need to schedule something to happen, for example, at same time next month or in 2.5 years. There is a simple way to obtain the corresponding timeout value that can be used in an event, statechart transition, Source or Delay object:

- **double toTimeout(int timeUnit, double amount)** – returns the amount of time (in model time units) from the current time to the (current time + a given number of days, months, years, etc.) with respect to daylight saving time. Use one of the constants **YEAR, MONTH, WEEK, DAY, HOUR, MINUTE, SECOND, MILLISECOND** as the **timeUnit**.

Because of *daylight saving time*, the time interval from 8 AM today to 8 AM tomorrow is not always 24 * hour() or 1 * day(). Therefore, if you wish, for example, an event to occur at 8 AM every day, you should set the recurrence time to **toTimeout(DAY, 1)**.

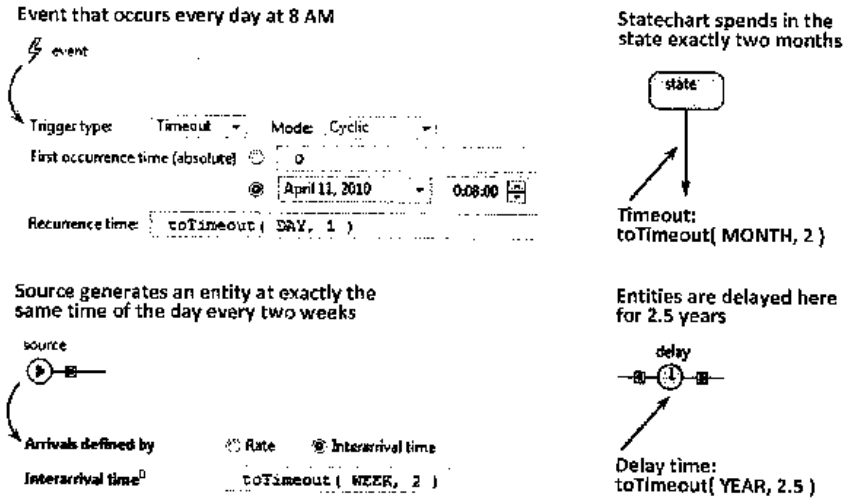


Figure 16.4 Using toTimeout() function in events, statecharts and Enterprise Library objects

16.3. Virtual and real-time execution modes

AnyLogic can execute the simulation model in two modes, *virtual time* and *real time* on a given scale. Virtual time is the "natural" execution mode when the simulation engine executes the model as fast as possible. The model time progresses unevenly and not continuously relative to real time; see Figure 16.5. In discrete event models, the model clock may instantly jump to the next event or may stall at one point while several simultaneous events are being executed. The model execution rate may appear more continuous if the model contains continuous-time dynamics (as in system dynamics models): in that case, the model is driven by the numeric solver, which makes small time steps that are more or less even. The computational complexity of events and equations obviously affects the speed of the execution of the model.

The virtual time mode is used when simulation performance is important and animation of the model dynamics is not needed, in particular in optimization, sensitivity analysis, parameter variation, Monte Carlo and other experiments where the model is run multiple times. System dynamics modelers also use the virtual time mode as they are typically interested more in the output graphs of the simulation than in the simulation process itself.

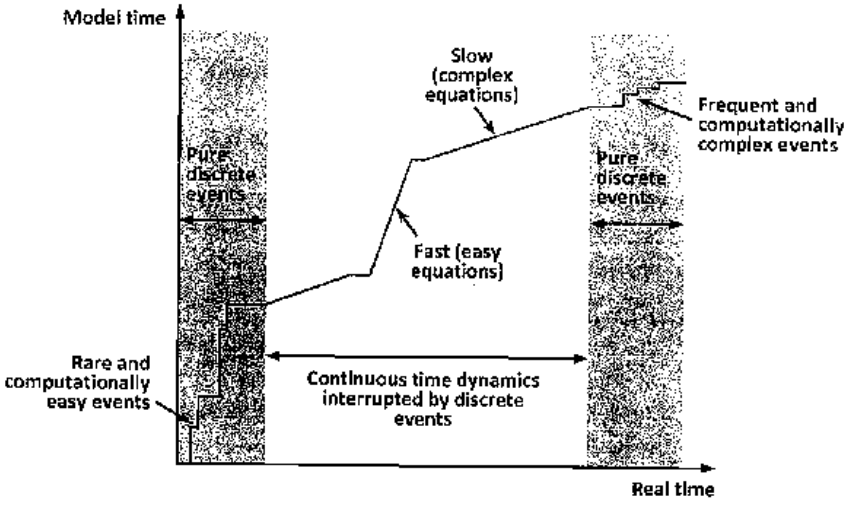


Figure 16.5 Virtual time ("natural") execution mode

In the scale to real-time mode, the engine tries to keep to a given scale, say 10 model time units (e.g., 10 simulated weeks) per 1 real second. If the model's computational complexity is not too high, the engine will periodically put itself in the "sleep" state

and wait for the correct real time to execute the next event or make the next step in the numeric calculations. Sometimes, though, the engine is unable to keep a given time scale because of too-frequent or too-complex events or because of a large system of equations and/or a too-small time step. Then the engine will work as fast as possible until it finds the next opportunity to maintain the real-time scale. Thus, the only thing the model can guarantee with respect to the real time is that the *model execution will never go faster than requested*; see Figure 16.6.

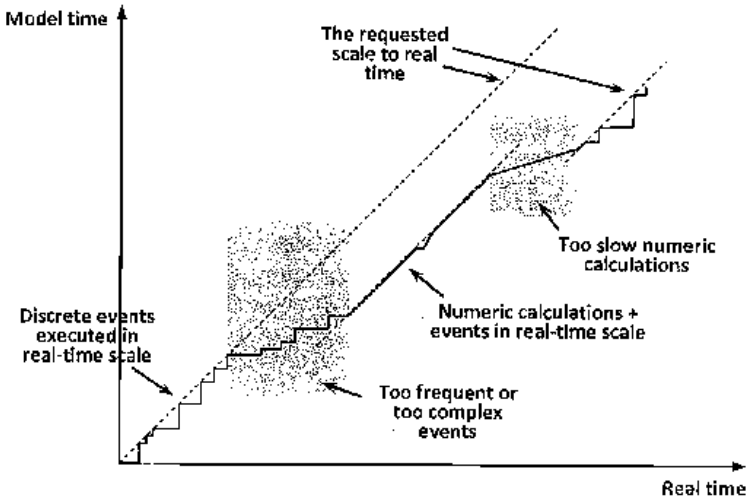

Model time
The requested scale to real time
Too slow numeric calculations
Numeric calculations + events in real-time scale
Discrete events executed in real-time scale
Too frequent or too complex events
Real time

Figure 16.6 Model execution in scale to real time

You can set the desired execution mode on a simulation experiment's **Presentation** property page (for other experiment types, the virtual time mode is assumed). Later on, you can use the model toolbar to change the mode during runtime or do it programmatically.

▶ **To set the default execution mode:**

1. Select the simulation experiment and open its **Presentation** property page.
2. Set the **Execution mode** either to **Virtual time** or to **Real time with scale**. The available scales range is from 1/512 to 512 model time units in one real second.

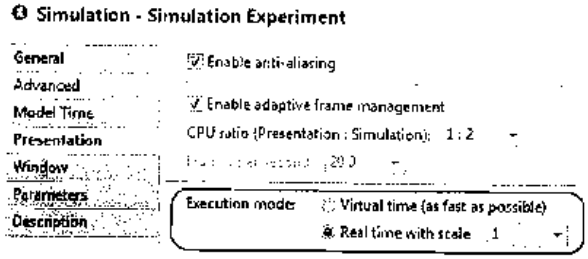


Figure 16.7 Setting the default execution mode

▶ **To change the execution mode in runtime using the model toolbar:**

1. Locate the **Time scale** section of the toolbar. If this section is not displayed, select **Time scale** in the toolbar customization menu.
2. Press to set the real-time mode with the default scale (i.e., the one set at the design time). Press or to slow down or speed up execution of the model. Press for the virtual time mode.

Execution mode API

For programmatic control of the execution mode, AnyLogic offers the following API:

- **setRealTimeMode(boolean on)** – sets the virtual (**on** is **false**) or real (**on** is **true**) mode of the model execution.
- **boolean getRealTimeMode()** – returns the current execution mode (**true** if real time, **false** if virtual time).
- **setRealTimeScale(double scale)** – sets the scale for the real-time mode (but does not change the current mode). The **scale** is the model time units per real second.
- **double getRealTimeScale()** – returns the current scale setting of the real-time mode.

In Chapter 8 you can find the example of programmatic control of the execution mode (Example 8.5: "Event slows down the simulation on a particular date").

References

Blender Foundation. (2013). *Blender* (Version 2.67) [Software]. Blender Foundation. Available from: http://www.blender.org

Boston Consulting Group. (n.d.). Retrieved June 1, 2012 from the Wiki: http://en.wikipedia.org/wiki/Boston_Consulting_Group

Compartmental models in epidemiology. (n.d.). Retrieved June 1, 2012 from the Wiki: http://en.wikipedia.org/wiki/Compartmental_models_in_epidemiology

Conway's Game of Life. (n.d.). Retrieved May 15, 2013 from the Wiki: http://en.wikipedia.org/wiki/Conway's_Game_of_Life

Geer Mountain Software Corporation (2002) *Stat::Fit* (Version 2) [Software]. Geer Mountain Software Corporation. Available from: http://www.geerms.com

Kelton D., Sadowski R. & Sturrock D. (2004). *Simulation with Arena*. 3rd ed . New York: McGraw.

Kolmogorov–Smirnov test. (n.d.). Retrieved June 1, 2012 from the Wiki: http://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test

MathWorks & NIST. (2012). *JAMA : A Java Matrix Package*. (Version 1.0.3). [Software]. Available from: http://math.nist.gov/javanumerics/jama/

New product development. (n.d.). Retrieved June 1, 2012 from the Wiki: http://en.wikipedia.org/wiki/New_product_development

Oracle. (2011). *Java™ Platform, Standard Edition 6. API Specification*. [Online]. Available from: http://docs.oracle.com/javase/6/docs/api/ [Accessed 3rd May 2013]

Oracle. (2013). *Control Flow Statements*. [Online]. Available from: http://docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html [Accessed 23rd May 2013]

QuickMBA.com. (2010). *Product Life Cycle* . [Online]. Available from: http://www.quickmba.com/marketing/product/lifecycle/ [Accessed 3rd May 2013]

Random number generator. (n.d.). Retrieved June 29, 2012 from the Wiki: http://en.wikipedia.org/wiki/Random_number_generator

Random .org. (1998-2012). *Random Integer Generator*. Available from: http://www.random.org/integers/ [Accessed 23rd May 2013]

Refsnes Data. (2013). *SQL Tutorial.* [Online]. Retrieved from: http://www.w3schools.com/sql/default.asp [Accessed 23rd May 2013]

Scale-free network. (n.d.). Retrieved June 27, 2012 from the Wiki: https://en.wikipedia.org/wiki/Scale-free_network

Small-world network. (n.d.). Retrieved June 27, 2012 from the Wiki: http://en.wikipedia.org/wiki/Small-world_network

Solo, K, & Paich, M. (2004). *A Modern Simulation Approach for Pharmaceutical Portfolio Management.* International Conference on Health Sciences Simulation (ICHSS'04), San Diego, California, USA. Retrieved from http://www.simnexus.com/SimNexus.PharmaPortfolio.pdf

Sterman, J. (2000). *Business dynamics : Systems thinking and modeling for a complex world.* New York: McGraw.

Sun Microsystems, Inc. (1999). *Code Conventions for the Java ™ Programming Language.* [Online]. Available from: http://www.oracle.com/technetwork/java/javase/documentation/codeconvtoc-136057.html [Accessed 23rd May 2013]

System Dynamics Society, Inc. (2013). System Dynamics Society [Online]. Available from: www.systemdynamics.org [Accessed 23rd May 2013]

The AnyLogic Company. (2013). *AnyLogic Help.* [Online]. Available from: http://www.anylogic.com/anylogic/help/

The Eclipse Foundation. (2013). Eclipse Classic. (Version4.2.2) [Software]. The Eclipse Foundation. Available from: http://www.eclipse.org/downloads/index.php

The Game of Life. (n.d.). Retrieved June 1, 2012 from the Wiki: http://en.wikipedia.org/wiki/The_Game_of_Life

Thomas Schelling. (n.d.). Retrieved June 11, 2013 from the Wiki: http://en.wikipedia.org/wiki/Thomas_Schelling

Trimble Navigation Limited & Google. (2013). *3D Warehouse.* [Online]. Available from: http://sketchup.google.com/3dwarehouse/ [Accessed 3rd June 2013]

United States Census Bureau. (2013) *United States Census Bureau.* [Online]. Available from: http://www.census.gov [Accessed 23rd May 2013]

UML state machine. (n.d.). Retrieved June 1, 2012 from the Wiki: http://en.wikipedia.org/wiki/UML_state_machine

Wallis, L., Paich M. & Borshchev A. (2004). *Agent Modeling of Hispanic Population Acculturation and Behavior.* The 3rd International Conference on Systems Thinking in Management (ICSTM 2004). Philadelphia, Pennsylvania, USA.

Web3D Consortium. (1999-2013). *Vivaty Studio*. (Version 1.0). [Software]. Web3D Consortium. Available from: http://www.web3d.org/realtime-3d/x3d-vrml/vivaty-studio-download

Web3D Consortium. (2013). *X3D Resources*. [Online]. Available from: http://www.web3d.org/x3d/content/examples/X3dResources.html [Accessed 23rd May 2013]

X3D. (n.d.). Retrieved June 5, 2012 from the Wiki: http://en.wikipedia.org/wiki/X3D

Index

Symbols

|, 398
!=, 398
&&, 398
||, 398
<, 398
<=, 398
==, 398
>, 398
>=, 398
3D animation, 539
3D objects, 549
3D Objects palette, 539
3D palette, 539
3D shapes, 542
3D Warehouse, 550
3D window, 552
  navigation, 541

A

Acceleration, 375
Action chart, 289
Active objects, 63
Active transition, 295
ActiveObjectArrayList, 407
ActiveObjectLinkedHashSet, 407
add(), 403
addFirst(), 404
addLast(), 404
Agent, 57
Agent based modeling, 49, 60
Aircraft maintenance checks, 314
Airport shuttle train, 372
Algebraic loop, 44
Alt+drag, 541
Ambient color, 560
Ambient light, 559
Analytical modeling, 27
Animation, 468
API, 392
  controls, 533
  dynamic events, 338
  events, 334
  shapes, 507
  statecharts, 306
Arc, 471
Arithmetic expressions, 396
ArrayList, 403, 405
Assignment, 412
Asynchronous discrete time, 54
Asynchronous time, 67
Attenuation, 560
Autodesk 3ds Max™, 27

B

Balancing feedback loop, 39
Base class, 384
Bass, F., 41
Blender™, 550
block, 417
Block comments, 419
boolean, 381
Boolean constants, 381
break statement, 414
Branch, 301
Button, 521, 523

C

C, 381
C++, 56, 381
Calendar, 592
Calibration, 42
Calibration experiment, 225
Callback, 378
Callout, 508
Carrying capacity, 208
case, 414

Case-sensitive, 408
Casting, 385
Causal loop diagram, 39
Checkbox, 521
Class, 60, 382
Class field, 387
Class variable, 387
Classification yard, 363
clear(), 404
Clipboard, 464
Code completion, 203, 391
Collada, 550
Color class, 516
Color interpolation, 518
Color picker, 514
Colors dialog, 514
Combo box, 522, 527
Command line arguments, 466
Comments in Java, 419
Compare Runs experiment, 221
Composite state, 288, 292
Computational RNGs, 583
Computer models, 27
Condition, 329
Condition event, 326
Conditional operator, 400
Connectivity palette, 431, 437, 451
Constant, 205
Constant complexity, 406
Constraints, 186
Constructor, 427
Constructor call, 383
Container object, 63
Continuous space, 86
Control, 521
  label, 524
  linking to value, 521
Control flow statements, 413
Control panel, 177
Conway, J., 76
Coordinate origin, 476, 483
Coupling rail cars, 363
Cruise speed, 353
CSV (comma separated values) file format, 435
Ctrl+drag, 477
Ctrl+Enter, 204
Ctrl+J, 395, 424
Ctrl+space, 203
Current velocity, 353
Curve, 470
Custom distribution, 570
Cyclic timeout event, 325

D

DAE file format, 550
Database object, 451, 453
Databases, 446
Date, 592
Date class, 594
date(), 390, 594
day(), 592
Daylight saving time, 596
Deceleration, 375
Decisio Consulting, 59
Decoupling rail cars, 363
Deep history state, 293
Default random number generator, 583
delay(), 219
DELETE statement, 461
Dependency link, 197
Deterministic model, 573
Differential equations, 41
Diffuse color, 560
Directional light, 559
Discrete event modeling, 321
Discrete event simulation, 321
Discrete space, 72
Discrete time, 48
Discrete uniform distribution, 569
Disease diffusion, 310
Distance-Based network, 111
Distribution fitting, 570
Distributions
  discrete uniform (uniform_discr), 569

exponential, 568
gamma, 568
normal, 568
Poisson, 569
random boolean (randomTrue, randomFalse), 568
triangular, 567
uniform, 567
double, 381
Drawing mode, 469
Dynamic event, 335
Dynamic histograms, 136
Dynamic properties, 488
Dynamic systems, 28

E

Edit box, 522, 525
Edit Points mode, 470
Embedded 3D presentation, 551
Embedded object, 63
Empirical distribution, 570
End of line comments, 419
Entity, 46
equals(), 398
Equations, 202
error(), 391
Escape sequences, 381
Euclidean neighborhood, 73
Euler, 44
Event, 324
Event queue, 323
Events, 321
instantaneous, 322
serialized, 322
Events view, 308, 338
Excel
reading data, 439
Excel file object, 437
Execution mode, 597
Exponential distribution, 568
Exponential growth, 212
exponential(), 390
Export to Forio™, 237
Export to runthemodel.com, 237
Exposed, 50
Extension, 471
External Java code, 429

F

Face point movement, 349
File chooser, 522, 529
Final state, 294
finishSimulation(), 391
Flight simulator, 177, 232
Flow, 41, 197
Flowchart, 289
For loop, 415
Forio™, 237
format(), 391
Forrester, J., 38
Frame, 497
Frame rate, 498
Fully connected network, 118
Function, 389
call, 412
definition, 394

G

Game, 232
Game of Life, 54
Gamma distribution, 568
get(), 403
get_Main(), 423
getEngine(), 391
getExperiment(), 391
getFirst(), 404
getIndex(), 391
getLast(), 404
getMinute(), 390
getOwner(), 391
getPresentation(), 391
getTimeUnit(), 591
Google Earth, 550

Google Sketchup Pro™, 550
Gordon, G., 45
GPSS, 45, 46
Grid, 476
Group, 480
Grouping shapes, 480
Guard, 300

H

Handling mouse clicks, 534
Harel, D., 287
HashSet, 405
Histogram, 370
Histogram data, 370
hour(), 592
HyperArray, 400

I

IBM, 27, 45
IF THEN ELSE, 204
if-then-else, 413
Image object, 472
Import, 429
Indentation, 418
Inherits, 384
Initial state pointer, 288
Inner classes, 428
Insert object, 451
INSERT statement, 461
Inspect window, 210
Instance, 60, 383
int, 381
Integer division, 396
Internal transition, 302
Inventory policy, 271
Investment policy, 272
isEmpty(), 403
isStateActive(), 392
iThink™, 45

J

JAMA package, 429
JAR files, 429
Java, 380
Java arrays, 401
Java class
  creation, 425
Java comments, 419
Java naming conventions, 408
Join, 449

K

Key-Value table object, 451
KML file format, 550
KMZ file format, 550
Kolmogorov-Smirnov test, 570

L

Length of stay, 370
Lexidyne, 59
Light, 559
Linear algebra package, 429
Linear complexity, 406
Linear Congruental Generator, 583
Link, 197
LinkedList, 404, 405
List box, 522
Little's law, 32
Local variable, 387
Locking shapes, 477
Locomotive, 377
log(), 390
Logarithmic complexity, 406
Logical operators, 398
  AND, 398
  NOT, 398
  OR, 398
Lookup table, 213
Loop, 198

M

M/G/1, 30
M/G/K, 33, 34
M/M/1, 30
M/M/K, 31
Matrices, 429
Matrix class, 429
Message passing, 128
Message queue, 299
Method, 37, 389
Method call, 383
min(), 390
minute(), 390, 592
MIT, 38
Model calibration, 42
Model log, 466
Model time, 589
Monte Carlo simulation, 573
Moore neighborhood, 73
MS Access, 446
MS Excel™, 27
MS SQL Server, 446
MS Visio™, 27
MS Word, 454
MTBF, 313
MTTR, 313
Multidimensional arrays, 403
MySQL, 446

N

Naming conventions, 408
Negative (balancing) feedback, 212
Negative feedback loop, 39
Neighborhood, 73
Networks, 110
Non-persistent, 480
Normal distribution, 568
NPD (New Product Development), 316
null, 383
Numeric constants, 381
Numerical methods, 44

O

Object, 383
On click, 535
Open-ended track, 348
Optimization, 182
  constraints, 186
OptQuest™ optimizer, 182
Oracle, 446
Oracle™, 380
Order of action execution, 303

P

pauseSimulation(), 391
PDF, 566
Performance, 259
Persistent, 480
Pharmaceutical NPD pipeline, 318
Physical RNGs, 583
Point light, 559
Poisson distribution, 569
Poisson stream, 30, 326
Pollaczek–Khinchine formula, 31
Polyline, 470
Positive (self-reinforcing) feedback, 212
Positive feedback loop, 39
pow(), 390
Power operation, 397
PowerSim™, 45, 195
Prepared statements, 462
Presentation, 468
Presentation palette, 469
Primitive data types, 381
Probability density function, 566
Probability distribution functions, 565
Procedural programming paradigm, 60
Process flowchart, 289
Process modeling, 321
Product delivery, 336
Product lifecycle, 273, 316
Product portfolio, 272
Progress bar, 522

Pseudo-random number generators, 583
Pseudo-state, 293
Purchase behavior, 311

Q

Q8 inventory policy, 271
Query object, 451
Queuing theory, 30

R

R&D, 272
Radio buttons, 522, 526, 530
Rail car, 377
Rail Library, 340
Rail topology, 341
Rail yard, 342
RailCar class, 377
RailYard, 342, 343, 344, 346, 347, 349, 379
Random boolean distribution, 568
Random network, 111
Random number generator, 583
Randomness, 573
  in agent based models, 578
  in process models, 574
  in system dynamics models, 581
Rate event, 326
READ mode, 431
Real time, 597
Reflection, 441
Reinforcing feedback loop, 39
Relational databases, 446
Remainder operation, 397
remove(), 403
removeFirst(), 404
removeLast(), 404
Replicated active object, 407
Replicated control, 532
Replicated object, 63, 132
Replicated shape, 498
Reproducible experiments, 585
Resource, 46
Resource with breakdowns, 313
return statement, 418
Ring Lattice network, 111
RNG, 583
Runge-Kutta, 44
RunTheModel.com, 237

S

Scale Free network, 111
Schelling, Th., 74
second(), 592
Seed, 583
SEIR model, 51
SELECT FROM statement, 448
Sensitivity Analysis experiment, 224, 285
setTimeUnit(), 591
Shadow variables, 199
Shallow history state, 293
Shape class, 514
Shape scaling center, 483
ShapeControl, 534
Shop working hours, 319
Short-circuiting behavior, 399
Shuttle train, 372
Simple state, 288, 290
Simulation method, 37
Simulation model, 29
Simulation modeling, 29
Simulation performance, 259
SIR model, 225
size(), 403
Sketchup 3D model, 550
Slider, 522
Small World network, 111
Snapping to grid, 476
Sorted collection, 406
Space types, 69
Specular color, 560
Spot light, 559
SQL (Structured Query Language), 446
SQL queries, 448
sqrt(), 390

S-shaped growth, 216
Standard output stream, 466
Start angle, 471
Start date, 593
Start time, 589
Stat::Fit, 570
State, 290
State diagram, 287
State information, 60
Statechart, 287
  API, 306
Statechart entry point, 291
Statement, 411
States, 287
Static event, 324
Static properties, 479
Statistics of the replicated object, 134
STELLA™, 45, 195
Stochastic model, 573
Stock, 41, 196
Stock and flow diagram, 41
Stock management game, 232
Stop date, 593
Stop time, 589
String, 381
String constants, 381
Subclass, 384
Sun™, 380
Super class, 384
Switch class, 349
switch statement, 414
Switches, 341
Synchronous discrete time, 54
Synchronous state transition, 304
Synchronous time, 67
System clipboard, 464
System dynamics, 38
System Dynamics palette, 196, 197, 198, 200
System.out.println(), 466

T

Table function, 213, 214, 433
Text file, 510
Text File object, 431
Text shape, 471
Texture, 515
Time complexity, 405
Time plot, 211
Time units, 590
time(), 390, 589
Timeout event that occurs once, 325
Token separators, 431
trace(), 466
traceln(), 391, 466
Track class, 347
Tracks, 341
Trailing point movement, 349
Train, 350, 375
Train class, 375
TrainCouple, 350
TrainDecouple, 350
TrainDispose, 350
TrainMoveTo, 350
TrainSource, 350
Transition, 287, 290
  guard, 300
  triggers, 295
Transparency, 517
TreeSet, 405
Triangular distribution, 566, 567
Trigger, 287
Truncation, 569
Type casting, 397

U

UML, 49
UML statecharts, 287
Uniform distribution, 567
uniform(), 390
Unit checking, 206, 220
United States Census Bureau, 519

Units, 206
Update object, 452
UPDATE statement, 461
User-controlled timeout event, 325

V

Variable declaration, 411
Vensim™, 45, 195, 569
Virtual time, 597
Vivaty Studio™, 550
VRML, 550

W

WebSphere Business Modeler™, 27
week(), 592
While loop, 416
Windows Notepad, 464
Wolfram Mathematica™, 27
WRITE mode, 431
WRITE_APPEND mode, 431

X

X3D file format, 550
XLS file format, 437
XLSX file format, 437

Z

Z-dimension, 539
Zoom, 476
Z-order, 474

ABOUT THE AUTHOR

Dr.Andrei Borshchev is the CEO and co-founder of The Anylogic Company,a leading provider of dynamic simulation tools,technologies,and consulting services for business applications,and is personally recognized as

one of the world's top experts in Agent Based ang multimethod simulation modeling.

Since 1998,Dr.Borshchev has been leading the development of Anylogic software enabling it to be considered the corporate standard for simulation in over 500 organizations worldwide.He has conducted hundreds of lectures,seminars,and training sessions on simulation modeling and Anylogic in the USA,Canada,Europe,Russia, Australia,China,Korea,Japan,and South Africa,and has also led numerous simulation based consulting projects across multiple markets including logistics,transportation, supply

chains,defense,healthcare,manufacturing,and consumer market analysis.

ABOUT THE BOOK

This is a practical guide to building simulation models.It explains how to choose the right constructs of the modeling language to create a representation of a real world system that is suitable for risk-free dynamic experiments.And this is the only book that comprehensively presents all three methods,or paradigms,in simulation modeling:Agent Based,System Dynamics,and Discrete Event.

The book is based on the modeling languages supported by Anylogic,the software tool that enables a modeler to utilize all three methods and to combine them in a single model. Written by the creator of the software,this book is the most in-depth information resource on Anylogic and multimethod modeling that currently exists. It covers topics from building simple Agent Based models with state charts,to designing interactive 3D animations,to developing data exchage with external programs.

With over 100 hans-on,step-by-step examples with different levels of complexity and various application areas,this book is extremely uesful to both simulation professionals and students learning simulation modeling.